Learning to Plan in LISP

John R. Anderson
Robert Farrell
Ron Sauers
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

This research was supported by the Personnel and Training Research Programs, Psychological Services Division, Office of Naval Research, under Contract No.: N00014-81-C-0335, Contract Authority Identification Number, NR No.: 157-465 to John Anderson.

unclassified
SECURITY CLASSIFICATION OF THIS PA(When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ONR-82-2	2. GOVT ACCESSION NO. AD-A121 736	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Learning to Plan in LISP		5. TYPE OF REPORT & PERIOD COVERED Interim report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John R. Anderson, Robert Farrell, Ron Sauers		8. CONTRACT OR GRANT NUMBER(s) N00014-81-C-0335
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 157-465
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel and Training Research Programs, Office of Naval Research, Arlington, VA 22217		12. REPORT DATE November 1, 1982
		13. NUMBER OF PAGES 46
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

production systems	computer simulation	knowledge compilation	retrieval
programming	analogy	proceduralization	planning
LISP	working memory	composition	
problem-solving	cognitive skill	problem decomposition	
goal structures	skill acquisition	recursion	automatic programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Protocols have been gathered of the first 30 hours of the learning of LISP. A simulation, GRAPES, has been developed that models the processes by which subjects write LISP functions to meet problem specifications. The GRAPES simulation is a goal-factored production system as specified in the ACT* theory (Anderson, 1983). The results are reported of the simulations of a number of problems and these are compared to the human protocols. GRAPES does simulate the top-down, depth-first flow of control exhibited by

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20) Abstract (con't)

subjects and produces code very similar to subject code. Special attention is given to modelling student solutions by analogy, how students learn from doing, and how failures of working memory affect the course of problem solving.

1. Introduction

We have been studying how novices learn to program in LISP. This is part of a more general goal of understanding how people learn complex skills (Anderson, 1982, 1983). A major part of the challenge and interest in this is that cognitive skills have complex control structures that must be learned. Computer programming is an excellent example of such a skill. The study of novices learning to program in LISP quickly gets us to the core of the problem of skill acquisition. Writing LISP functions involves a complicated control structure and it is the kind of control structure novices have never dealt with. Please note that "control structure" refers here to the control structures governing the programming behavior not control structures in the program.

1.1. The Data Base

We have looked extensively at the first 30 hours of novice programming behavior of three subjects (SS, WC, and BR). In these protocols, subjects studied a text on LISP - SS studied Siklossy (1976), WC studied Winston (1977), and BR studied Winston & Horn (1981). We recorded their verbal protocols, kept their paperwork, and kept a record of their terminal interactions. The individual session varied from 45 minutes to two and a half hours, depending on what seemed to be natural units and natural breaking points. Approximately one quarter of the session time was spent reading and discussing the text; the other three-quarters of the time was spent doing various exercises. The subject worked with an experimenter who tried to do as little teaching as possible and to let the student learn from the text. The main responsibility of the experimenter was to query the subject about what they were thinking, why they tried various solutions, etc. However, if the subject had a serious misunderstanding or was lost in the problems, the experimenter would intervene with tutorial assistance.

We feel that we have a pretty good record of the learning that was occurring in these sessions. Subjects were instructed not to think about LISP when they were not in the experimental session. It seemed that it was easy for them to comply with this request. They were also not permitted to keep the textbook between sessions.

While the 30 hour protocols from these subjects has been the major source of data for theory construction, we have also looked at protocols from these subjects much later after they had continued their LISP education and also looked at protocols from relatively advanced LISP programmers. In addition, we have assigned various LISP problems to a large class learning LISP. While we cannot get from this class source any information about the real-time problem-solving it does provide information about the distribution of final solutions. This provides one basis for judging the representativeness of the solutions we see from our three subjects.

1.2. The GRAPES Simulation

We developed GRAPES (a goal-restricted production system) to model subject problem solving in the context of writing a LISP function to calculate an input-output specification. This program is an interpreter for a set of production rules that write LISP code in a top-down manner. Each production has a condition which specifies a particular programming goal and various problem specifications The action of the production can be to embellish the problem specification, write or change LISP code, or to set new subgoals. Representative examples of such productions[1] are:

```
    IF the goal is to add List1 and List2
THEN write (APPEND List1 List2)

    IF the goal is to check that a recursive call to a function will terminate
       and the recursive call is in the context of a MAP function
THEN set as a subgoal to establish that the list provided to the MAP function
            will always become NIL after some number of recursive calls
```

We feel that the GRAPES simulations do a good job of reproducing the essential aspects of the protocols that we observe. However, it is never the case that a perfect correspondence is obtained. We could, of course, account for any perturbation in the data by introducing an ad hoc rule that would produce just that perturbation. However, we constrained ourselves to rules which we felt reasonable to suppose the subject had acquired. We hope to eventually be able to explain the perturbations on a principled basis. However, what is described in this paper is only an approximation to such a final theoretical account. This fits in well with the view that development of scientific theory is an approximating series in which each subsequent theory gives a better fit to the data.

1.3. Relation to ACT*

The GRAPES architecture is just a specialization of the general ACT* architecture (Anderson, 1983) to achieve an efficient simulation of that architecture in the context of LISP programming. Like ACT*, it involves a dichotomy between general declarative knowledge represented in the form of a semantic network and procedural knowledge represented in production form. ACT* involves a sub-theory about how declarative facts are stored and retrieved and a sub-theory about how productions are matched, executed, and acquired. GRAPES incorporates the procedural sub-theory but does not incorporate the declarative sub-theory (only for reasons of efficiency and simplicity). In GRAPES declarative facts are simply stored and retrieved without error. GRAPES makes particularly heavy use of the ACT* proposal for using goals to guide the matching and acquisition of productions. In ACT* behavior is organized hierarchically according to a set of goals such that higher-level goals decompose into lower-level goals. In ACT* it is also possible for productions to execute that respond to the data without any goal specifications. However, to date we have not used such data-driven productions in simulating LISP programming behavior.

In ACT* there are two basic mechanisms for simulating production learning. The first, knowledge compilation, involves building production system rules that summarize the essential product of the computation of a set of rules. The second, knowledge tuning, involves adjusting the conditions of productions to make them more appropriate in their range of application. We have implemented a knowledge compilation mechanism within the GRAPES system and will describe its application later in the paper. While a tuning process is also important in modelling learning to program, this has not yet been developed in GRAPES.

1.4. Overview

The remainder of the paper will fall into four major sections. The first three will report on three simulation efforts. With these sections as "data", the final section will draw a set of significant conclusions about the nature of programming in LISP and the acquisition of this skill.

2. REACHABLE

The first problem we would like to discuss is the REACHABLE problem, which was borrowed from Barstow (1979). It is a problem too advanced to be attempted by any of our subjects in their first 30 hours of learning and so we only have protocols from more advanced programmers solving it. It serves to demonstrate the overall competence of the simulation program in terms of its ability to simulate the solutions to relatively complex problems. The REACHABLE problem is also interesting in that it tends to evoke a wide variety of different final solutions from subjects. Thus, it will serve as a test of whether the same GRAPES control structure is capable of explaining the variety of programming behavior that can be observed.

We will look at GRAPES reproducing two rather different solutions by two subjects. The individual solutions and simulations contain a number of interesting features that will serve to support some of the conclusions in the final section of the paper. In these simulations, we set as our goal to account for the coding behavior of our subjects in terms of the actual lines of code they wrote. While we tried to reproduce certain aspects of the protocol, there was no attempt to have a step for step simulation of the subject. This goal will be attempted for the later problems in this paper.

The REACHABLE problem, as it was specified to subjects, is illustrated in Figure 1. The program is supposed to operate on a directed graph like the one at the top of the figure. According to the specification of the problem by Barstow (1979), the function is to be given three arguments - a start node, a list of nodes in the graph, and a list encoding of the graph structure. The task is to find all nodes reachable from the start node following directed arrows. Most subjects did not use the second argument, the list of graph nodes, and some expressed puzzlement that it was included since it seemed redundant with the graph encoding. The graph encoding is a list of lists where each sublist gives the connections of one of the nodes. The first element in that sublist is the node itself and the second element is the list of nodes connected to that node. The exact example given in Figure 1 was used in explaining REACHABLE to all subjects.

Insert Figure 1 about here

2.1. Simulation of Subject WC

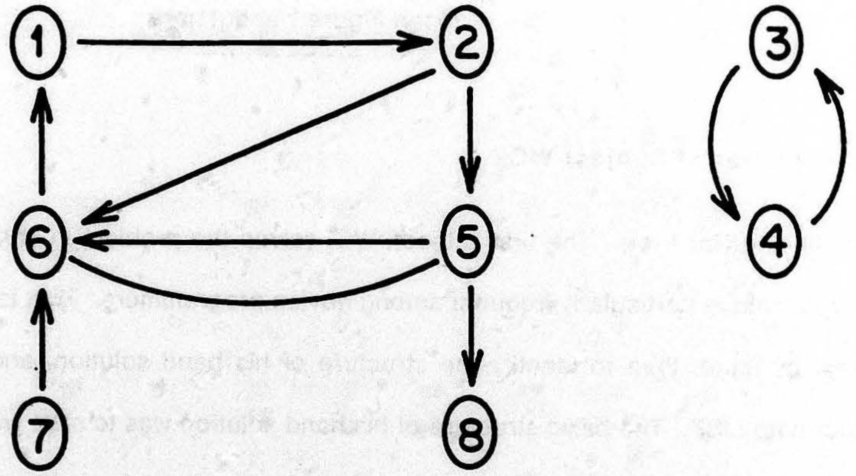
2.1.1. The Initial Plan. The first subject, WC, solved the problem in about two hours and used a type of technique particularly frequent among novice programmers. That is, he tried to first solve the problem by hand, then to identify the structure of his hand solution, and then to map that hand solution onto LISP. The basic structure of his hand solution was to start with the start node; retrieve connected nodes from it; add these to the answer; scan the answer for a node he had not tried; if there was such a node, add its connected nodes to the answer; if not, terminate with the answer. The following is a particularly relevant portion of his protocol:

> "One thing that occurs to me is to just start wandering through the network and keep track of where I am. So I start at one, the start node, and say let's go look at the graph subset of one".

Our subject tried to map these steps into operations on list structures. Figure 2a illustrates the hierarchical plan that we think he developed for performing operations on list structures. The plan began by initializing the answer list to the start node. Then he had a chunk of behavior organized by a control construct that we call <u>repeat-until failure</u>. This involves performing a set of operations until a test results in a failure. We believe that this is a natural pre-programming control construct for most subjects. It has an obvious relationship to the repeat-until-success construct that was an essential part of the TOTE hierarchies of Miller, Galanter, & Pribram (1960).

Insert Figure 2 about here

REACHABLE



ARGUMENTS

START: 1

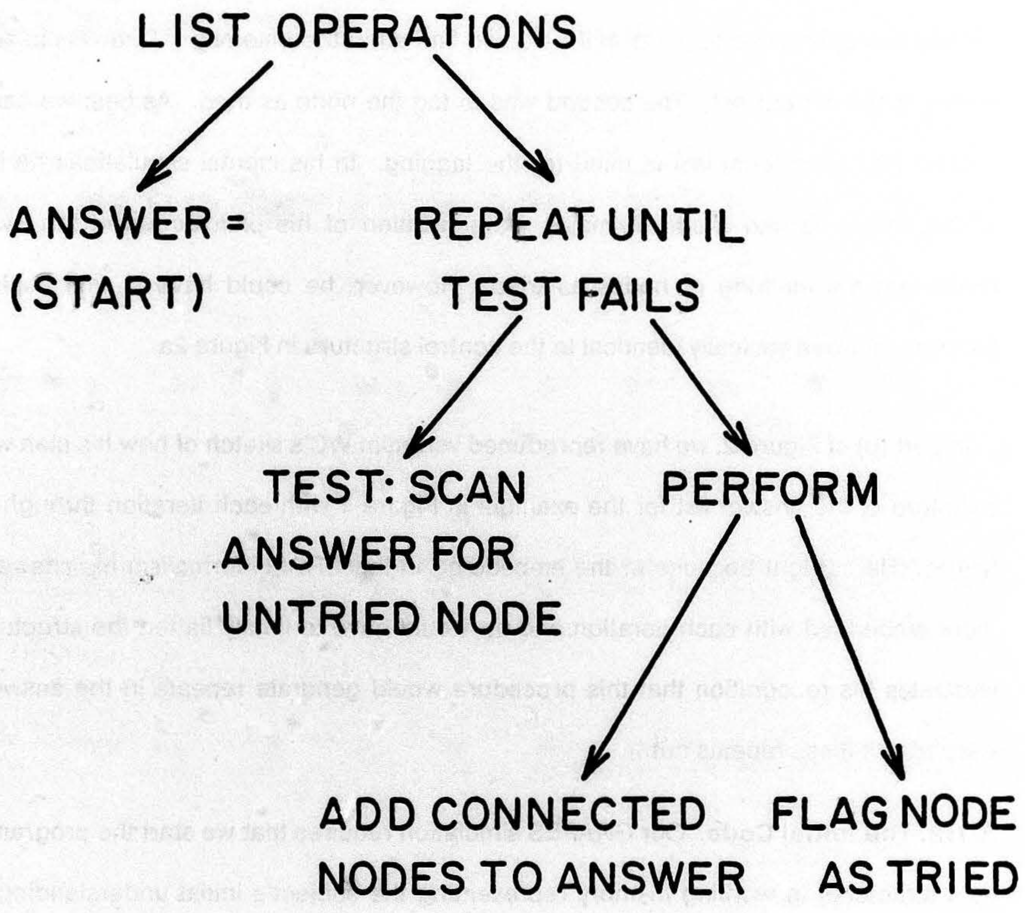
NODES: (1 2 3 4 5 6 7 8)

GRAPH: ((1 (2)) (2 (5 6))
(3 (4)) (4 (3))
(5 (6 8)) (6 (1 5))
(7 (6)))

ANSWER: (1 2 5 6 8)

FIGURE 1

(a)



(b)

```
(1)
(1)(2))
(1  (2 (5 6))
(1  (2 ((5 (6 8))  6))
(1  (2 ((5 (6 8)) (6 (1 5)))
(1  (2 ((5 (6 8]
```

FIGURE 2

The test in the repeat-until-failure loop involved determining whether there was an untried node on the answer list. Our subject thought of implementing this vaguely in terms of a scan of the answer list. He had two operations to perform if he could find an untried member. One was to add its connected nodes to the answer list. The second was to tag the node as tried. As best we can determine, our subject had no mechanism in mind for the tagging. In his mental simulations he just remembered which nodes he had tried. A major reorganization of his plan occurred later when he tried to implement the marking of nodes as tried. However, he could have in fact implemented a LISP function that was basically identical to the control structure in Figure 2a.

In part (b) of Figure 2, we have reproduced verbatim WC's sketch of how his plan would change the structure of the answer list for the example in Figure 1 with each iteration through the repeat-until-failure. He thought because of the embedding in the GRAPH formalism his answer would become more embedded with each iteration and he would have to finally flatten the structure. The last line illustrates his recognition that this procedure would generate repeats in the answer and he would have to edit these repeats out.

2.1.2. The Initial Code. Our GRAPES simulation requires that we start the program off with a set of data structures in working memory representing the subject's initial understanding of the problem. Then various productions can be matched to the contents of working memory and they will start generating the LISP code. To simulate WC on this problem we put into working memory a specification of the input-output relation calculated by the function and the hierarchical plan in Figure 2a.[2] The first production to apply responds to the existence of this hand solution and sets as a subgoal to map it onto a LISP function. The next rule to apply is the default rule for mapping:

```
    IF the goal is to map a procedure onto LISP
THEN map its subprocedures
```

The two main procedures of the hand solution are the initialization of the answer list and the repeat-until-failure. Therefore, it sets two subgoals to map these onto LISP. The code it puts out at this point is:

```
(def reachable
   (lambda (node graph list)
      <?>
      <?>))
```

It has printed out the standard template for a LISP definition. This corresponds to the almost universal practice of our subjects who will whip out the basic definition format before doing anything else or thinking carefully about how they will write the LISP code. The two <?>'s are GRAPES place-holders for the code to achieve the two major subgoals of the plan.

GRAPES flow of control proceeds in a depth-first, left-to-right manner. Therefore, the program next focuses on writing the code to initialize the answer list. The next two production rules to apply are:

```
    IF the goal is to create a structure
THEN create a variable and use SETQ

    IF the goal is to make a list
THEN use LIST
```

And the LISP code they create is:

```
(def reachable
   (lambda (node graph list)
      (prog (@List2)
         (setq @List2 (list node))
         (return <?> ))))
```

The first production results in the creation of a PROG structure to permit a local variable @List2 to hold the answer. The second production recognizes that a list containing the start node can be created with the LISP function LIST. In the context of a PROG a RETURN is also inserted.

Now attention focuses on the goal of mapping the second part of the hand plan - the repeat-until-failure. The next production to apply is:

```
    IF the goal is to map a repeat-until-failure
THEN create a COND in a loop and give the COND two clauses where
            one clause performs the test and the other clause deals with failure of the test.
```

It recognizes that a repeat-until-failure can be achieved by a two-clause COND structure in a loop.

The first clause will perform the test and apply if the test is successful. The second clause will deal with the case when the test fails. The code at this point is:

```
(def reachable
   (lambda (node graph list)
     (prog (@List2)
       (setq @List2 (list node))
    loop (cond (<?> <?>) (t (return <?>)))
     (go loop))))
```

2.1.3. Reorganization of the COND Clause. GRAPES now focuses on coding the condition for the first COND clause. This is where the program brings itself to face the issue of how it is going to implement the tagging of the node as tried. The subject hit this same issue at this point. At this point in time the following rule applies

```
    IF the goal is to find an item on a list with a property
THEN code a test for that property
          and then search the list linearly for an item with that property
```

This sets the plan of a linear search of a test and makes the immediate goal deciding how to code the test used in that search. Because a test for <u>not tried</u> is negative, the following rule applies:

```
    IF the goal is to test if something does not have a property
THEN use NOT and set as a subgoal to test whether the item has the property
```

Then the following rule applies

```
    IF the goal is to test if an item has a property
THEN create a list which will be updated with all items that have that property
          and test whether the item in question is a member of that list.
```

The program at this point has planned the code (NOT (MEMBER <?> @list4)) where <?> will be expanded to code the item and @list4 is the answer test. It now turns to its goal of applying this code in a scan through the list:

```
    IF the goal is to search a list linearly for an item that satisfies a test
       and a set of operations are to be performed on the item
THEN create a two-clause COND structure in a LOOP
       where the first COND clause tests for an empty list and returns the answer
       and the second COND clause tests for the property and performs the operations
       and the list is reset to its CDR after the COND
```

The code at this point is illustrated below:

```
(def reachable
   (lambda (node graph list)
      (prog (@List2 @List4)
         (setq @List2 (list node))
      loop (cond((not @List2)(return @List4))
               ((not (member <?> @List4)) <?>))
            (setq@list2 (cdr @list2))
           (go loop ))))
```

The original COND structure is replaced by this new COND structure.

2.1.4. Finishing the COND Clause. The next thing GRAPES focuses on is how it will code the element in the unfinished condition for the second COND clause. It recognizes that the element is the first member of @list2 and the following basic rule applies:

```
    IF the goal is to code the first element of a list
THEN use CAR
```

Attention now focuses on coding the action for the second COND clause. This action corresponds to the operations in Figure 2a to be performed on untried items. Therefore, the following rule applies:

```
    IF the goal is to map a hand plan
       and the hand-plan performed a set of operations
THEN set as subgoals to map these hand operations
```

This creates the goals of mapping the operations of adding the connected nodes to the possibility list and tagging the element as tried. The following four productions[3] apply in sequence to code the operations of adding connected nodes to the possibility list:

```
    IF the goal is to reset the value of a variable
THEN use SETQ

    IF the goal is to add two lists together
THEN use APPEND

    IF the goal is to get the second member of a list
THEN use CADR

    IF the goal is to retrieve a sublist that begins with a key
THEN use ASSOC
```

The program then focuses on the subgoal of coding the tagging of the item as tried. The following two rules apply in order:

```
    IF the goal is to give an item a property
```

```
          and there is a list to be updated with  all items with that  have that property
     THEN add the item to the list with the property

          IF one wants to add an item to a list
     THEN use CONS
```

The final code is:

```
(def reachable
    (lambda (node graph list)
      (prog (@List2 @List4)
          (setq @List2 (list node))
       loop (cond ((not @List2) (return @List4))
                  ((not (member (car @List2) @List4))
                   (setq @List2
                      (append @List2 (cadr (assoc (car @List2) graph))))
                   (setq @List4 (cons(car @List2) @List4))))
                 (setq @List2 (cdr @List 2))
              (go loop))))
```

One thing that this example obviously illustrates is that a lot of specific rules underlies the generation of code in LISP. One of the basic claims we will be making about learning in LISP is that it consists in a major part of the acquisition of many such special purpose rules.

Below is the LISP code produced by WC for this problem:

```
(def reachable
   (lambda (list graph node)
     (prog (@List4 @List2)
         (setq @List4 (list node))
         (setq @List2 (cadr (assoc node graph)))
    loop (cond ((not @List2)(return @List4))
               ((member (car @List2) @List4)
                 (setq @List2 (cdr @List2)))
               (t (setq @List2 (append @List2(cadr
                      (assoc (car @List2)graph))))
                 (setq @List4 (cons (car @List2) @List4))
                 (setq @List2 (cdr @List2))))
         (go loop))))
```

I have renamed the variables in the subject's program to correspond to those used by GRAPES. Also the subject wrote a helping function rather than the CADR-ASSOC composition. There are three significant differences in the final code. First, the subject initializes @List2 and @List4 with the results of the first iteration through the loop in the GRAPES program. Second, he has encoded the

NOT-MEMBER predicate by making it the default T clause after a MEMBER test. Third, the resetting of @list2 occurs within each clause. Each of these differences could have been eliminated by introducing specific production rules or variants in existing rules. However, such tuning seemed pointless and we wanted to use the same GRAPES rules for a variety of problems by a variety of subjects. In any case, the degree of correspondence is quite good given the wide range of solutions that we have seen for REACHABLE. The reason the solutions are similar is that GRAPES and the subject begin with the same working memory state.

2.2. Simulation of Subject 2

In contrast to subject WC, subject 2 did not use a hand simulation but rather worked from a definition of the REACHABLE relationship. He worked from the following definition which he articulated

```
Node Y is reachable from node X if
   (a) There is a direct path from X to Y
or (b) There is a node Z such that
     (i) There is a direct path from X to Z
     (ii) Node Y is reachable from Z.
```

The subject attributed his ability to formulate this definition to prior exposure to graph theory. In simulating this subject, we loaded a representation of this definition in working memory together with the same specification of the problem as we gave to WC. We ran this simulation with the same rule set that we used on the first simulation. This will provide a test of whether the rules were especially tuned to reproduce WC's solution. In what follows we will star the rules that were also used in the prior simulation.

2.2.1. The Initial Coding The first three rules to apply are given below in order:

```
R1:      IF the goal is to code the answer
            and the answer is defined as all members
                with property A or property B
         THEN set as subgoals to
                1. code all the members with property A
                2. code all the members with property B
            and UNION these together

R2:*     IF the goal is to get the second member of a list
```

```
        THEN use CADR

R3:*        IF the goal is to retrieve a sublist that begins with a key
        THEN use ASSOC
```

The code at this point in time is given below:

```
(def reachable
   (lambda (node graph list)
     (union (cadr (assoc node graph)) <?>)))
```

The first rule translates the OR in the definition of REACHABLE into a UNION. The next two rules we saw apply in the previous simulation and in this context they encode the directly connected part of the REACHABLE definition.

The program now turns to coding the distantly reachable nodes (part b of the definition). Then the following rules apply:

```
R4:         IF the goal is to obtain all the elements which have a relation
                    to any member of a list
        THEN use MAPCONC on that list with a function that will return all the
                    elements that have a relation to an argument

R5:         IF the goal is to code a relation
               and that relation has been coded earlier
        THEN create a variable
               and set it to the value of the earlier coding
               and use it in the current coding

R6:         IF the goal is to code a function to achieve a relation
               and the function is to be called by a function that is trying to achieve the same relation
               but their arguments are different
        THEN create a recursive call to the function
               and set as a subgoal to check that the recursion will terminate
```

The code that results at this point in time is given below:

```
(def reachable
 (lambda (node graph list)
  (@function1 node)))

(def @function1
   (lambda (node)
      (prog (@LVar1)
        (return
        (union (setq @LVar1 (cadr (assoc node graph)))
            (mapconc '@function1 @LVar1))))))
```

Production R4 responded to the definition of distantly reachable nodes by planning to map a function that obtained the reachable nodes through a list of the immediately reachable. Production R5 recognized that the immediately reachable had already been calculated and set a local variable LVar1 to these. Production R6 recognized that the function to be mapped was identical to the current one and created a recursive call. It is at this point that the simulation splits the calling function, reachable, from the recursive function, @function1, which will just take one argument. This is done to enable the simple recursive call in the MAP context.

2.2.2. Terminating the Recursion The subgoal that is set by this last function is to guarantee that the recursion will terminate. At this point the following two rules apply to refine this goal into something that can be achieved.

```
R7:         IF the goal is to check that a recursive call to a function will terminate
               and the recursive call is in the context of a MAP function
         THEN set as a subgoal to establish that the list provided to the
                     MAP function will always become NIL

R8:         IF the goal is to establish that a list will always become NIL
               and it contains part of the answer to a problem
               and the full answer is finite
         THEN set a subgoal to avoid repeating elements that
                     are already part of the answer
```

Thus, the program reasons to the conclusion that it does not want to repeat answers in the @LVar1 structures that it calculates. The following rule recognizes a technique for achieving this

```
R9:         IF the goal is to avoid repeating elements on a list of answers
         THEN create a global list to hold the answers so far
               and set as subgoals
                     1. to edit the list each time it is created so that it does not contain
                           elements in the global list
                     2. to update the global list each time new answers are obtained
```

A variable @GVar1 is created to serve this function of a global variable. Since it only has to be global to the recursive function @function1, it is made a PROG variable in the calling function REACHABLE. The following rule applies to edit out repeats when the variable @LVar1 is calculated

```
R10:        IF the goal is to find all the members in one list that are not in another
```

THEN use LDIFFERENCE

The function at this point in time is:

```
(def @function1
   (lambda (node)
      (prog (@LVar1)
         (setq @GVar1 <?>)
         (return
         (union (setq @LVar1 (ldifference (cadr (assoc node graph))
                        @GVar1))
             (mapconc '@function1 @LVar1))))))
```

The remaining goal is to code the updating of the global variable so that it includes the new nodes calculated as the value of @LVar1. However, @LVar1 is calculated after the intended updating of the global variable. Therefore, the following rule applies

R11: IF the argument needed is embedded inside some later LISP code
THEN extract the code and bring it to where it is needed

Then the following rule applies to encode the updating of @GVar1.

R12:* IF the goal is to add two lists together
THEN use APPEND

The final LISP code for @function1 is:

```
(def @function1
   (lambda (node)
     (prog (@LVar1)
        (setq @LVar1 (ldifference (cadr (assoc node graph)) @GVar1))
        (setq @GVar1 (append  @LVar1  @GVar1))
        (return (union @LVar1 (mapconc '@function1 @LVar1))))))
```

In contrast to the previous GRAPES simulation where the correlation between subject code and program code was partial, this GRAPES simulation precisely reproduced the code of the subject (after renaming of variables). The solution to this problem is quite different from the first, but was produced by the same GRAPES program with just a different initial working memory. Thus, the REACHABLE problem can evoke a wide range of behavior from GRAPES and from subjects.

Despite these considerable differences, the two protocols produced by GRAPES have an important feature in common with each other and with the protocols of all the subjects we have observed on the

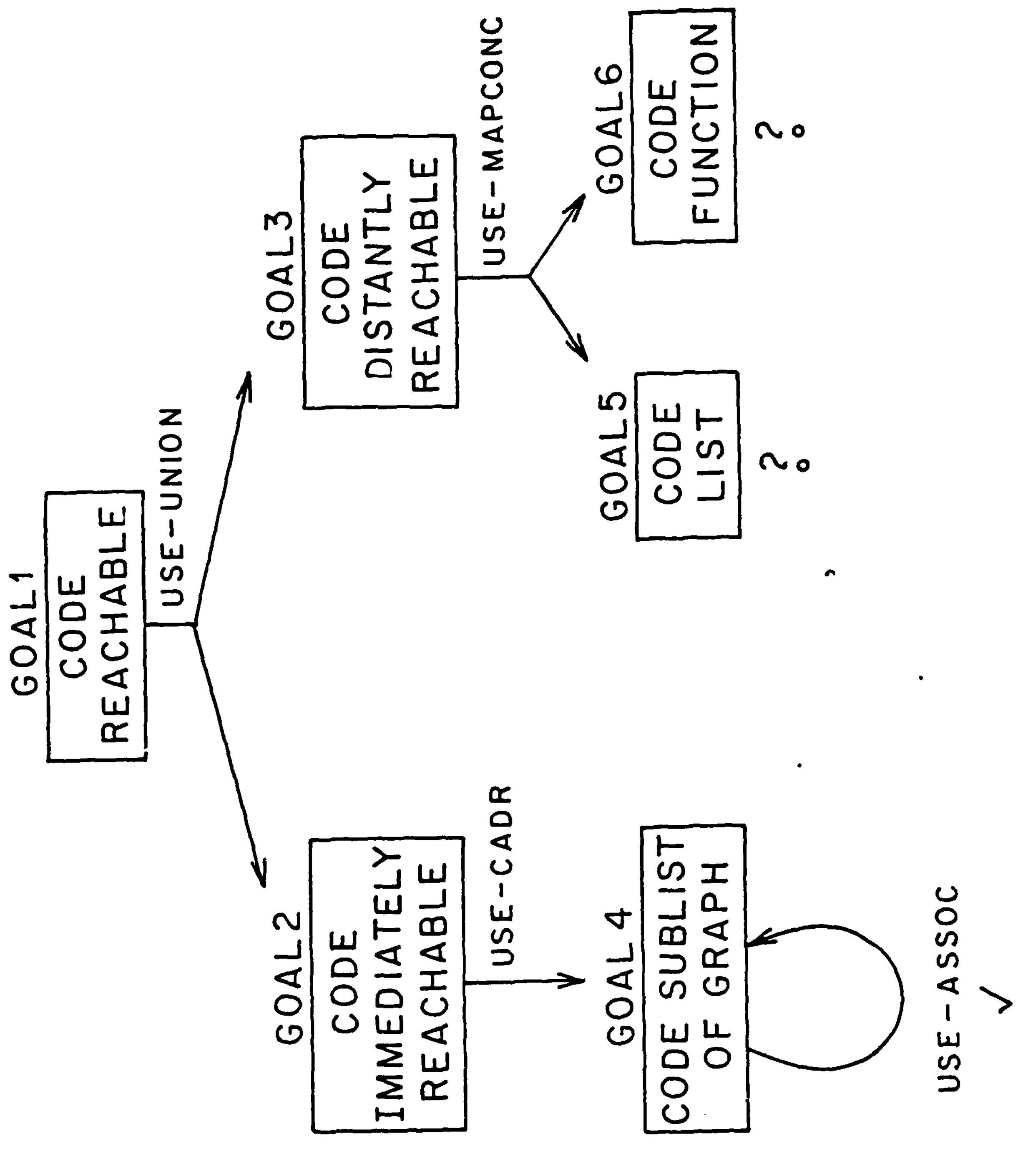
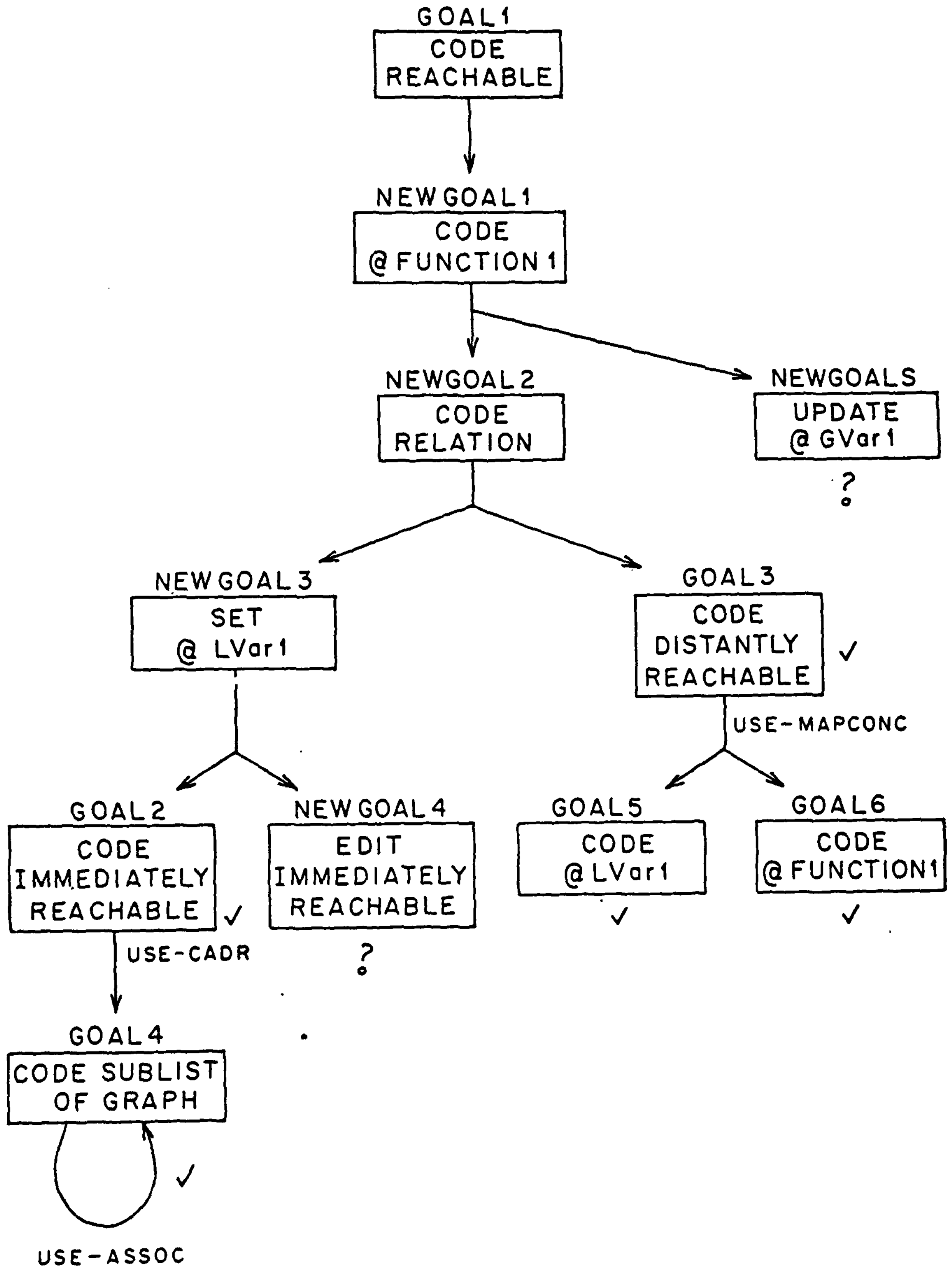

GOAL1
CODE REACHABLE
USE-UNION
GOAL2
CODE IMMEDIATELY REACHABLE
GOAL3
CODE DISTANTLY REACHABLE
USE-CADR
USE-MAPCONC
GOAL 4
CODE SUBLIST OF GRAPH
USE-ASSOC
✓
GOAL5
CODE LIST
?
GOAL6
CODE FUNCTION
?

FIGURE 3



GOAL1
CODE REACHABLE
NEWGOAL1
CODE @FUNCTION1
NEWGOAL2
CODE RELATION
NEWGOALS
UPDATE @GVar1
?
NEWGOAL3
SET @ LVar1
GOAL3
CODE DISTANTLY REACHABLE
✓
USE-MAPCONC
GOAL2
CODE IMMEDIATELY REACHABLE
✓
NEWGOAL4
EDIT IMMEDIATELY REACHABLE
?
GOAL5
CODE @LVar1
✓
GOAL6
CODE @FUNCTION1
✓
USE-CADR
GOAL4
CODE SUBLIST OF GRAPH
✓
USE-ASSOC

FIGURE 4

problems. This is that they involve a reorganization of the LISP function. For subject 1, this occurred when he revised the COND structure. For subject 2, it occurred when he introduced global variables to keep track of tried nodes. The reorganizations for all subjects are concerned with the problem of avoiding looping when searching the graph. Many of the subjects, including these two, were aware of the danger of looping from the outset, but simply could not anticipate how it would impact on the problem solution until they had gotten some distance into their coding.

2.3. The Goal Structure

The previous examples illustrated how performance is controlled through the interaction of a large number of rules. These rules are organized according to a hierarchical goal structure. The various rules respond to goals and can create subgoals. So in subject 2's protocol after rule R4 the goal structure in Figure 3 had been created. The program generates this goal structure in a left-to-right depth-first manner. The code is written in this same order. In fact the hierarchical structure of the goals often is linked into the hierarchical structure of the code.

Insert Figures 3 and 4 about here

At this point there are two open goals indicated by question-marks at the terminal nodes in Figure 3. These are the goals of coding the list which the function is supposed to be mapped through and of coding the function itself. Rules R5-R9, which respond to these goals, do not just unpack these into further subgoals. Rather they transform the original goal structure. The goal structure after their application is illustrated in Figure 4. The recursive function @function1 for use in MAPCONC has been created and inserted between the REACHABLE function and the use of UNION. The code of SETQ has been inserted before the coding of the immediately reachable. Open goals are the editing of the @LVar1 (the newly connected nodes) and the updating of @GVar1 (the list of already connected nodes). GRAPES selects the left-most open goal to work on, which in this case involves editing the local variable. In achieving the remaining open goals, one further transformation of the goal structure is produced. This involves taking the structure under the SETQ for @LVar1 and

```
(POWERSET '(A B C)
  = ((A B C) (A B) (A C) (B C) (A) (B) (C) ())
```

FIGURE 5

moving it up in the goal tree to before where @GVar1 is set. Note that order in the goal tree does not necessarily always correspond to order in the code. There is an example of non-correspondence in Figure 4. The function UNION was coded before the updating of the global variable. This is quite typical in programming behavior where subjects will code later lines first and then insert the earlier lines. Another example of this occurred in the simulation of subject 1 where the code (not (member <?> @List4)) was decided before the COND structure was created into which it was inserted.

The hierarchical generation of goal structure and subsequent transformation of these structures is similar in character to the conception of planning in Sacerdoti (1977). Anderson (1983) has argued that this kind of control structure is quite ubiquitous in cognitive skill. For instance, it is found in natural language where phrase structure is the sign of the hierarchical structure and transformational structure of language is the sign of goal transformations.

As these examples make clear, a program develops out of a series of successive reworkings of an initial plan. This is very much like the successive refinement model of algorithm design advocated by Kant and Newell (1982) -- although their formalism for representing plans differs considerably from ours.

3. POWERSET

The POWERSET example occurs towards the end of our 30 hour learning sessions and is the most demanding of the problems that we present to our subjects during the 30 hour learning. It differs from REACHABLE in that it tends to evoke the same solution from all subjects. Thus, the coding behavior of GRAPES corresponds quite closely to all subjects at the level of lines written. However, typical subject behavior contains numerous errors, recoveries, and false starts not seen in our simulation. After discussing the program's behavior on POWERSET, we will discuss some of the errors made by one typical subject on this problem.

Insert Figure 5 about here

The POWERSET problem as it is presented to subjects is illustrated in Figure 5. The subject is told that a list of atoms encodes a set of elements and he is to calculate the powerset of that set -- that is, the list of all sublists of the original list, including the original list and NIL. Each subject was given an example of what the POWERSET was for a three element list. The three subjects we observed spent from under two hours to over four hours solving this problem. In each case, they spent about one-third of their time uncovering a key insight and the other two-thirds of their time working out the LISP code that would capitalize on this insight.

We have also assigned this problem to a number of programming classes and gathered informal problem solution reports. There are two types of solutions which subjects are prone to attempt and which tend to distract them from the correct insight:

1. There is a strong tendency to try to implement the way they would solve the problem by hand. For most subjects this hand solution is one in which they calculate the null list, then all the singleton lists, then all the doubleton lists, etc.

2. Some subjects are distracted by the fact that certain sublists can be achieved quite easily by taking CDR's. So, given the example (A B C), (B C), (C), and ( ) can be gotten by taking successive CDR's. This leaves the difficult task of calculating the non-CDR's.

The essential insight is illustrated in Figure 6. This involves noticing the relationship between the POWERSET on the full list and POWERSET on the tail (CDR) of the list. In Figure 6 we denote by X the result of POWERSET on the full list and we denote by Y the result of POWERSET on the tail of the list. Subjects noted that Y provided half of the members they would need for X. Second, they noted that the other half could be gotten from Y by adding A, the first member of the list L, to each number of Y. Thus, X is formed from the lists Y and Z, where Z is formed from Y by adding the first member of L to each member of Y.

Insert Figure 6 about here

```
L= (A B C)

X= (POWERSET L)                 Y= (POWERSET (CDR L))
  = ((A B C)                      = ((B C)
     (A B)                           (B)
     (A C)                           (C)
     (A)                             ())
     (B C)
     (B)
     (C)
     ())
X= Y+Z            WHERE           Z= ((A B C)
                                      (A B)
                                      (A C)
                                      (A))
```

Z IS FORMED FROM Y BY ADDING A TO EACH MEMBER OF Y.

FIGURE 6

The decision to consider the relationship between (POWERSET L) and (POWERSET (CDR L)) is not just a stab in the dark. It is dictated by a recursive programming technique that the students were taught called tail-recursion or CDR-recursion. This technique involves assuming that the function will return the correct result for the CDR of the list and trying to use this result to calculate the correct answer for the whole list.

3.1. GRAPES Problem Solution

Figure 7 illustrates GRAPES' goal structure for this problem. GRAPES keys off the fact that the argument is a list to attempt the CDR-recursion technique. This technique involves two subgoals. One is to write the code for the recursive step and the other is to write the code for the terminating step which is when the argument to POWERSET is the empty list, NIL. Under the recursive step, there are two subgoals. One is to characterize the relationship between POWERSET of the full list and POWERSET of the tail of the list. The other is to convert that characterization into LISP code.

Insert Figure 7 about here

As it is not our concern to model the pattern-matching abilities and set-theoretic knowledge that permit this insight into the recursive relationship, we basically provided GRAPES with a representation of this relationship. With the relationship identified, GRAPES then turns to the goal of converting that into LISP code. The answer, X, is the sum of two lists, Y and Z. GRAPES recognizes that this can be achieved by using the LISP function APPEND with the two arguments, Y and Z. It therefore sets as subgoals to calculate Y and Z. It recognizes that under the assumption of CDR-recursion, it can use the recursive call (POWERSET (CDR L)) to calculate the value for Y.

Then GRAPES turns to coding the second argument, Z. Z is formed from Y by adding A to each member of Z. GRAPES sets out to write a new function ADDTO that will form this second argument. The goal structure for the working of ADDTO is illustrated in Figure 8. The function is written with to the same cdr-recursion technique as is POWERSET. More advanced students might recognize this

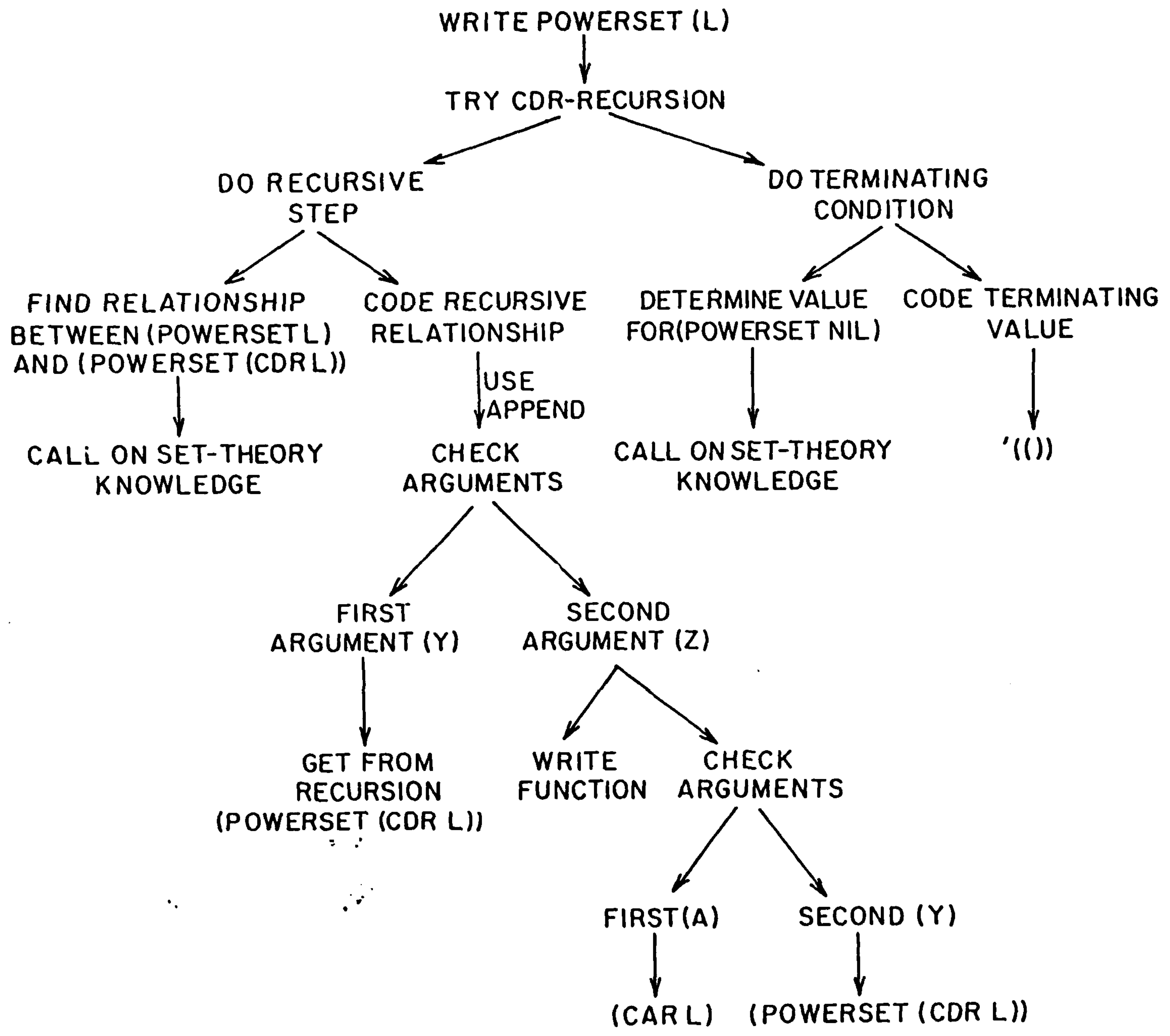

WRITE POWERSET (L)
TRY CDR-RECURSION
DO RECURSIVE STEP
DO TERMINATING CONDITION
FIND RELATIONSHIP BETWEEN (POWERSET L) AND (POWERSET (CDR L))
CODE RECURSIVE RELATIONSHIP
DETERMINE VALUE FOR(POWERSET NIL)
CODE TERMINATING VALUE
CALL ON SET-THEORY KNOWLEDGE
USE APPEND
CHECK ARGUMENTS
CALL ON SET-THEORY KNOWLEDGE
'(())
FIRST ARGUMENT (Y)
SECOND ARGUMENT (Z)
GET FROM RECURSION (POWERSET (CDR L))
WRITE FUNCTION
CHECK ARGUMENTS
FIRST(A)
SECOND (Y)
(CAR L)
(POWERSET (CDR L))

FIGURE 7

as basically a simple iterative structure and solve it according to a PROG or MAP structure, but we are simulating LISP students at the point where they have not been taught about PROG's or MAP's and only know about recursion within LISP and not iteration. The final code written by GRAPES for ADDTO is given below:

```
(def addto
   (lambda (a y)
      (cond ((not y) nil)
         (t (cons (cons a ( car y))
            (addto a (cdr y)))))))
```

Insert Figure 8 about here

The program returns from writing ADDTO and its further behavior is illustrated in Figure 7. It next decides how to calculate the arguments to ADDTO. After this it turns to coding the terminating condition for POWERSET. Here it calls upon its set theory knowledge to determine that the powerset of the empty set is a set containing the empty set. Given the coding principles of this problem, it translates this into '(()). Its final code is:

```
(def powerset
   (lambda (l)
       (cond ((not l) '(()))
        (t (append (powerset (cdr l))
         (addto (car l) (powerset (cdr l))))
          ))))
```

3.2. Comparison to WC

As illustrated above, given the insight, the coding of POWERSET provides very little difficulty for GRAPES. It can be coded directly from recursive relationship in a straightforward hierarchical manner. It does not involve the complex transformation that we saw with REACHABLE. However, individual subjects spent from 1 to 3 hours converting this insight into code. While the overall structure of their protocols is like that of the simulation and their final functions certainly are similar, these subjects spend their time making errors of coding and recovering from these errors.

The subject WC's protocol is easiest to discuss because it involves the fewest errors. Like the

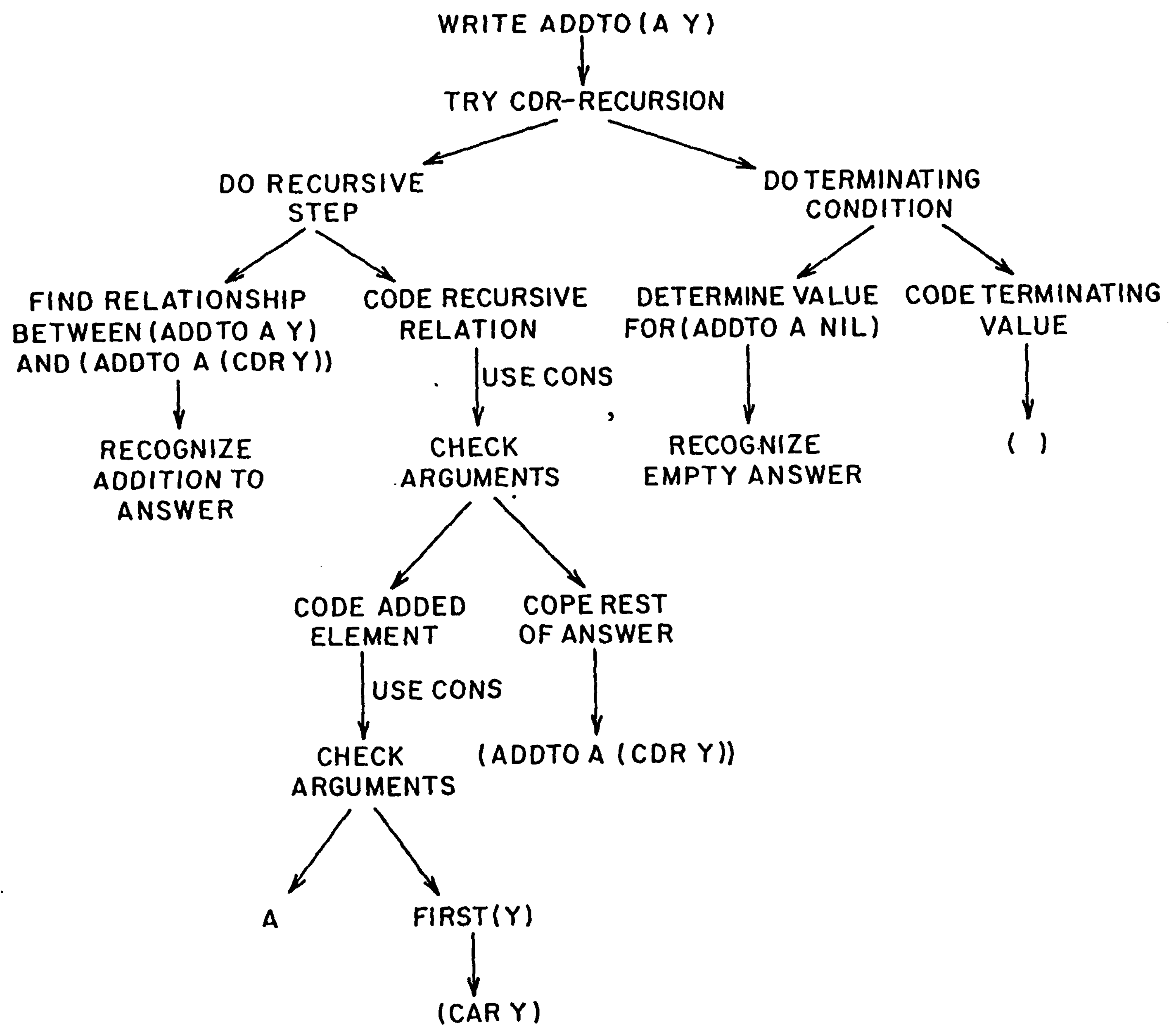

WRITE ADDTO (A Y)
TRY CDR-RECURSION
DO RECURSIVE STEP
DO TERMINATING CONDITION
FIND RELATIONSHIP BETWEEN (ADDTO A Y) AND (ADDTO A (CDR Y))
CODE RECURSIVE RELATION
DETERMINE VALUE FOR (ADDTO A NIL)
CODE TERMINATING VALUE
USE CONS
RECOGNIZE ADDITION TO ANSWER
CHECK ARGUMENTS
RECOGNIZE EMPTY ANSWER
( )
CODE ADDED ELEMENT
COPE REST OF ANSWER
USE CONS
CHECK ARGUMENTS
(ADDTO A (CDR Y))
A
FIRST (Y)
(CAR Y)

FIGURE 8

program, he consciously applied CDR-recursion as a strategy, discovered the relationship between (POWERSET L) and (POWERSET (CDR L)), and clearly articulated to himself the relationship that we denote X = Y + Z -- specifically, that the answer was the sum of <u>two</u> lists. He then turned to coding the relationship. Despite the fact that he had seen the relationship as the sum of two lists, he first turned to trying to encode just Z. Thus, he misremembered the relationship. After a minute thinking about it, he spontaneously corrected himself and recognized that he would have to APPEND two lists together. Like GRAPES, he realized that one argument to APPEND, Y, could be coded as (POWERSET (CDR L)).

He then turned to writing the code for Z and his first code was (UNION (CAR L) (POWERSET (CDR L))). UNION is a function which combines two lists and avoids repeats. This clearly will not give Z. It seems he has a vague specification in working memory of combining A with Y and UNION matches this specification on the basis of it being a combining function. WC knows quite well what UNION does. As evidence of this, he corrects his code a couple of minutes later, and articulates what is wrong without intervention of the experimenter.

Like the GRAPES simulation, WC does not have the concept of an iterative operation as distinct from tail recursion and so analyzes ADDTO as another case of tail recursion. When he first turned to coding the iteration or recursion step and he wrote (CONS (LIST A (CAR L))(ADDTO A CDR L))). This differs from the correct code in that the function LIST is used rather than a CONS. Rather than combining A and (B C) to get (A B C), this will combine them to get (A (B C)). Once again we see on our subject's part the confusion of two similar functions--in this case LIST, which makes its arguments elements of a list, is confused with CONS, which adds its first argument to the list which is its second argument. This is all the more interesting because this line of code also contains a correct use of CONS. It needs to be stressed that upon reflection, WC knows quite well the distinction between CONS and LIST. Again it is the matter of sloppy retrieval in the course of problem solving.

Then WC turned to writing the appropriate code for the terminating condition--i.e., when ADDTO is called with arguments A and NIL. His first thought is that he should add A to this empty list and return (A). That is, he has lost sight of the fact that the second argument to ADDTO is a list of lists and he should add A to each sublist. This is another example of the subject losing track of what it is that he had intended to do. The subject discovered the problem with this code by mental simulation and put in the correct terminating value, namely, NIL.

At this point, we typed the function definition into the terminal and tried it out on some sample problems. By tracing the function, he spotted and diagnosed the problem caused by his use of LIST rather than CONS. He changed this and the function ran correctly. It should be noted that WC corrected this problem without help from the experimenter and without looking up CONS or LIST in his text.

Having completed ADDTO, he then returned to writing POWERSET. His first remark was "Now, why did I write ADDTO?" He had completely forgotten the series of goals that led to this. He had to re-read the code he had written to reconstruct his goals. Thus, GRAPES clearly differs from WC in that it has perfect memory for the goal structure in Figure 7.

After he reconstructed his plan for POWERSET, WC turned to planning the terminating condition. His first inclination was to return NIL as the value when POWERSET was called with the argument NIL. This was the only place that we intervened with some suggestions. We pointed out that the powerset is defined as the set of all subsets of a set. A set itself is considered a subset of itself. Therefore, the set itself was among the sets in the powerset of a set. Therefore, among the elements of the powerset of the empty set should be the empty set itself. Thus, the result for POWERSET of NIL should be (NIL) or (()). WC was completely unconvinced by this argument but obediently returned (NIL) as the result in the terminating condition.

Then we typed the function into the terminal and watched it run with a TRACE on POWERSET. When he saw POWERSET return (NIL) for the value of NIL and when he was how this result was used by higher levels of POWERSET, he remarked that he now understood why (NIL) was the right value for the terminating condition. He still did not understand our logical argument but he had a procedural understanding of why the result was essential to the correct working of the function.

There is a close correspondence between WC and GRAPES in the overall flow of control among goals created by the decomposition strategy. However, there are frequent failures of memory on WC's part which are not part of the simulation. He both loses track of partial products calculated in the course of planning a function and incorrectly retrieves functions from memory. It needs to be emphasized here that WC is a very intelligent and capable person. So these errors are information about the nature of being a novice in LISP programming and not about WC. We have observed a similar high frequency of errors in all our novice subjects. Such errors are less frequent with advanced LISP programmers. It is also noteworthy that errors like the LIST-CONS confusion are almost non-existent when subjects are asked to execute a command at the top-level of LISP. They only come out embedded in the context of a problem with considerable working memory load. (A recent experiment conducted on a class of 60 novice programmers has confirmed that LIST-CONS confusions are more common when the function use is embedded within the other functions). A clear implication of this is that a major difference between the current implementation of GRAPES and our subjects is working memory capacity.

3.3. Analysis of Retrieval Failures

Working memory failures are clearly the cause of certain problems in the protocol like (1) forgetting that the answer to REACHABLE was a sum (Y + Z) of lists; (2) forgetting that the argument to ADDTO was a list of lists; (3) forgetting why ADDTO was written. We think working memory failures are also responsible for the incorrect retrievals of functions like UNION and LIST.

The following is our analysis of the LIST-CONS confusion. It is similar to what Norman (1981) called a description error. We assume that the subject represents as his goal

1. To create a LIST L
2. where the first element of L is A

3. and where the rest of the list consists of B

This matches the specification of CONS. On the other hand, if the third clause above had consists replaced by contains, then it would match the specifications of LIST. If we assume that the relation contains is simpler than consists and involves a subset of its semantic features, we would predict that subjects would tend to lose the distinguishing features under heavy memory load and retrieve LIST instead of CONS. Also, this analysis would predict that CONS should not be intruded instead of LIST. This asymmetry is clearly the case in our protocols. The asymmetry has been shown to be statistically reliable in large-scale class experiments as well.[4] This analysis is also consistent with a different CONS ~~LIST-CONS~~ error that we will see in the next protocol.

It is interesting to speculate why there should be this contains-consists confusion and why contains should be more primitive. In standard LISP conventions consists would be mapped directly into tail and contains would be defined indirectly in terms of tail. However, this is apparently not the way the novice programmer thinks of it. Perhaps, the novice's thinking is dominated by the marks that denote list structure, particularly the parentheses. The representation of "the rest of X contains Y" might be represented as "X = . . .Y") or "Y occurs just before the right parentheses of X". The representation of "the rest of X consists of Y" might be represented as "X = Z) where Y = (Z)" or "Z is what occurs between the left and right parentheses of Y and Z occurs just before the right parentheses of X". The underlined part in the above representation is the extra information which distinguishes consists from contains. If this information were lost or misrepresented in working memory then LIST would provide the best matching pattern to the goal specification.[5] A somewhat different way of putting the point is as follows: In the definition of LIST the argument Y is a proper subpart on the answer X. In contrast, the argument is not a proper subset of the answer in the definition of CONS. It is cognitively simpler when the answer preserves the structure of the argument.

4. ONETWO

In the previous protocols, we contented ourselves with having GRAPES model the overall flow of control and the correct codings of the subject. In part this was because the complexity of the protocols makes a detailed simulation extremely burdensome and perhaps impossible. In the simulation of the ONETWO protocol by subject SS we aspired for a much more detailed level of correspondence. This protocol is just over an hour in length. It occurred at the sixth hour of the learning protocol. It is also interesting because it contains within it an example of a significant act of learning. This involves the acquisition of new problem-solving operator.

The ONETWO problem required the subject to write a function which would take a list as an argument and return a new list consisting of the first two elements of the argument list. The LISP functions that the subject knew at this time included CONS but the subject had not yet learned about LIST. She knew about CAR and CDR and with these she had defined functions that would return the first, second, and third arguments of a list. These were the only functions that she had written up to this point in time.

4.1. Initial Attempt at ONETWO

She flailed at writing the function ONETWO and the experimenter suggested writing a simpler function, ADDTWO, which would take two arguments and make a list out of them. This problem she was able to make some headway on. It is interesting to speculate why ADDTWO was more tractable than ONETWO. As we will see, the basic problem and its solution did not change in going from ONETWO to ADDTWO. However, by reducing the complexity of the task by one level, the burden on working memory was reduced so that the subject was better able to match operators.

Figures 9-14 illustrate the simulation's attempts to solve ONETWO. Given the perfect correspondence between the simulation and SS's protocols, we infer that these figures also describe the goal structures that were guiding her problem solutions.

Insert Figure 9 about here

Figure 9 illustrates the first work that was done on the ADDTWO subproblem. The first operator

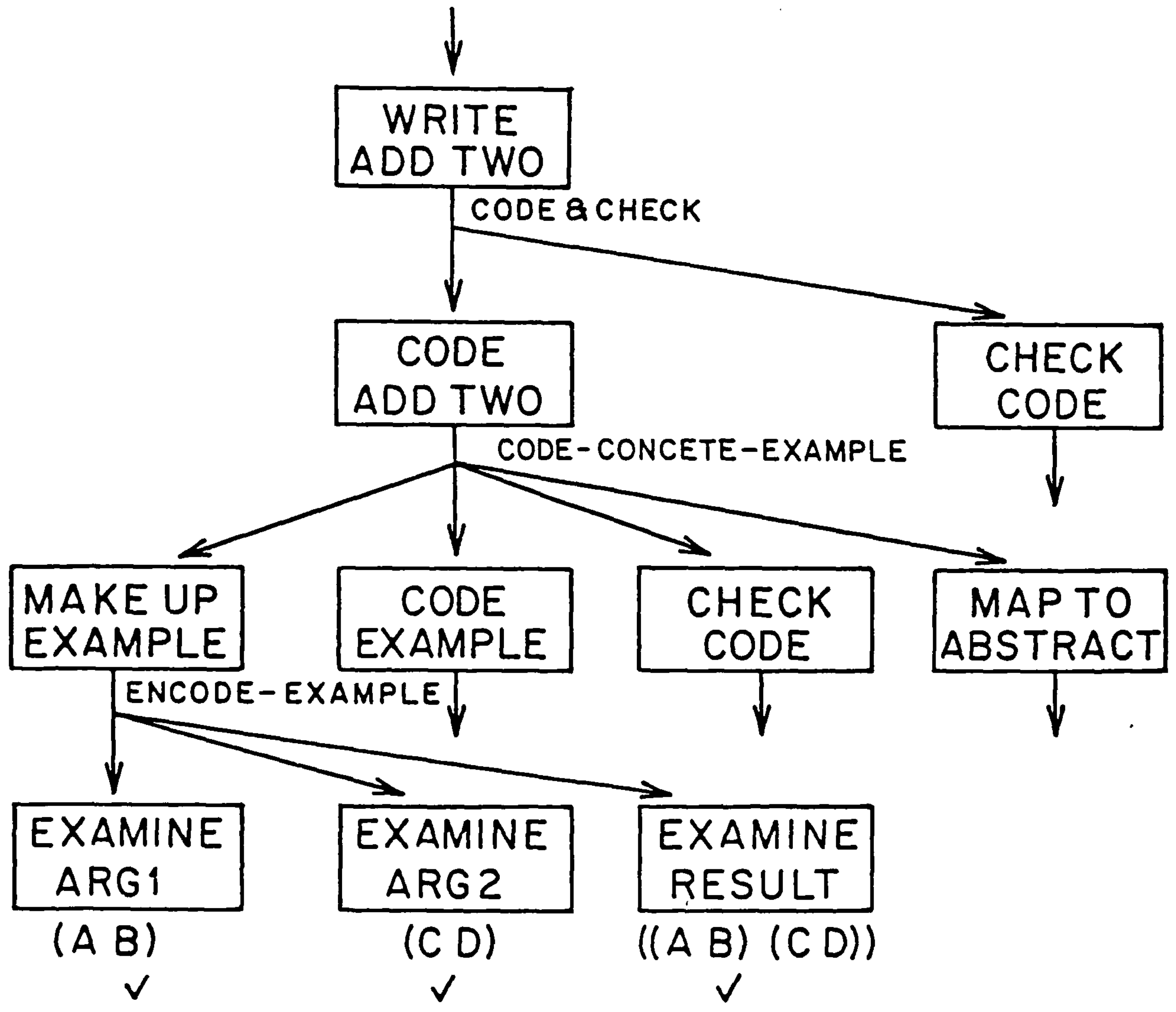

WRITE ADD TWO
CODE & CHECK
CODE ADD TWO
CHECK CODE
CODE-CONCETE-EXAMPLE
MAKE UP EXAMPLE
CODE EXAMPLE
CHECK CODE
MAP TO ABSTRACT
ENCODE-EXAMPLE
EXAMINE ARG1
EXAMINE ARG2
EXAMINE RESULT
(A B)
✓
(C D)
✓
((A B) (C D))
✓

FIGURE 9

decomposes this into the subgoals of coding the function and checking the code. The first operator set subgoals to come up with concrete examples of the input to ADDTWO and what its output should be, to find some code that could be typed at the top level that would convert the concrete input into the concrete output, to check this code, and then to map this code into an abstract function. The inputs she chose to pass to ADDTWO were (A B) and (C D). Why she chose list arguments we are unsure. The result she wanted for these inputs was ((A B) (C D)).

Figure 10 illustrates the process by which she decided how to create this example at the top level. After deciding on the example, she went through an episode where she explicitly reviewed the definition of all the functions she knew, searching for an appropriate one. She selected CONS. We represented the definition of CONS to GRAPES as

> The first argument of CONS is any S-expression and the second argument is a list. Its result is a list. The first element of the result is the first argument. The rest of the result consists of the second argument.

She and GRAPES choose CONS on the basis of the fact that a list was wanted and CONS makes lists. Having selected CONS the subgoals were now to determine what arguments to pass to CONS in order to get the intended result. Note that this is a different use of CONS than in the previous simulations. Previously, GRAPES knew both what the inputs and the result were and selected CONS because it would map one onto the other. Here CONS is chosen solely on the basis of its result and it is necessary to decide what arguments to pass to it.

Insert Figure 10 about here

The critical piece of information in selecting the first argument is the definition statement <u>The first element of the result is the first argument.</u> GRAPES interfaces this with the desired result, ((A B) (C D)), to determine that the correct argument should be (A B).

Next, SS and GRAPES turn to the second argument. The appropriate part of this definition is <u>The rest of the result consists of the second argument.</u> Matching this would retrieve ((C D)) as the second argument. However, our subject retrieved (C D). We assume, in line with our discussion of the

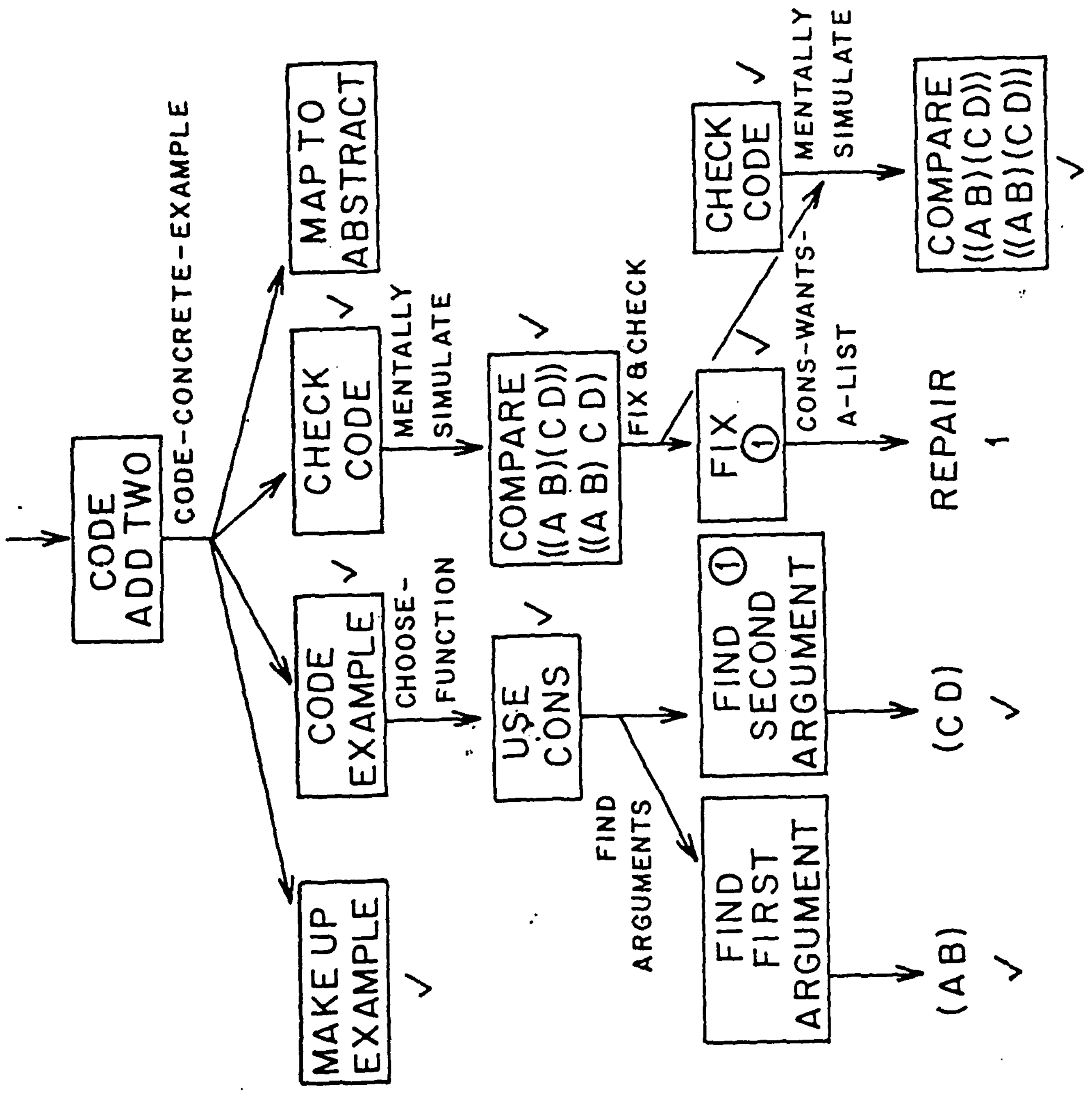

CODE ADD TWO
CODE-CONCRETE-EXAMPLE
MAKE UP EXAMPLE ✓
CODE EXAMPLE ✓
CHECK CODE ✓
MAP TO ABSTRACT
CHOOSE-FUNCTION
MENTALLY SIMULATE
USE CONS ✓
COMPARE ((A B)(C D)) ((A B) C D) ✓
FIND ARGUMENTS
FIX&CHECK
FIND FIRST ARGUMENT
FIND SECOND ARGUMENT 1
FIX 1 ✓
CHECK CODE ✓
CONS-WANTS-A-LIST
MENTALLY SIMULATE
(A B) ✓
(C D) ✓
REPAIR 1
COMPARE ((A B)(C D)) ((A B)(C D)) ✓

FIGURE 10

LIST-CONS confusion, that the semantic features of consists were partially lost and this statement became The rest of the result contains the second argument. Thus, the same error that causes LIST to be retrieved rather than CONS will produce this error. We manipulated GRAPES' working memory so that it would produce this error.

The subject and GRAPES mentally simulated what the outcome would be of the code (CONS '(A B) '(C D)). This involved retrieving the definition of CONS again. As evidence that her definition of CONS was not in error, she correctly determined that ((A B) C D) would result as an answer. This corresponded to an error she had encountered frequently and we assume she had compiled an operator to repair this which embedded the second argument to CONS in an extra list. In this way, she and GRAPES recover from their error and make up the concrete example (CONS '(A B) '((C D))).

This concrete example is different than the concrete example in the first REACHABLE simulation, but it is serving a similar function. In REACHABLE the subject had solved a problem by hand and used the structure of the hand solution to guide the LISP code. Here the subject has actually created some LISP code that can be typed into the top level of LISP and is going to use the structure of this code to guide the creation of an abstract LISP function. In neither case is the mapping from concrete to abstract trivial.

4.2. The Mapping

Insert Figure 11 about here

Figure 11 illustrates the simulation of SS's initial attempt to map from the concrete code to an abstract LISP function. First she maps CONS in the concrete code into CONS in the LISP function. At this point the structure of the function is

```
        (def addtwo (lambda (one two)
            (cons <?> <?>)))
```

The remaining task is to map the two concrete arguments into abstract arguments. She first focuses on mapping (A B). The following rule applies:

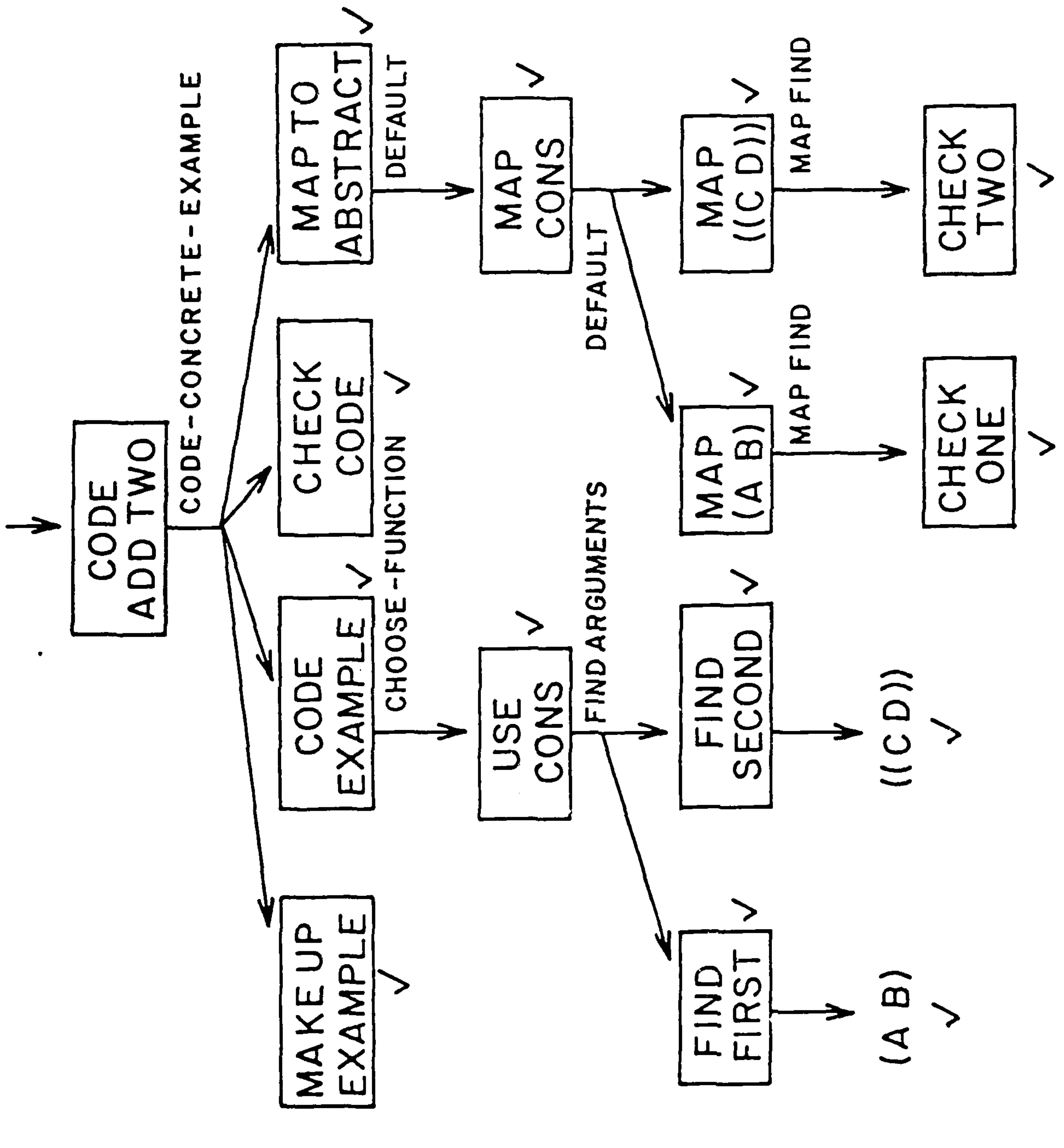

CODE ADD TWO
CODE-CONCRETE-EXAMPLE
MAKE UP EXAMPLE
CODE EXAMPLE
CHECK CODE
MAP TO ABSTRACT
CHOOSE-FUNCTION
DEFAULT
USE CONS
MAP CONS
FIND ARGUMENTS
DEFAULT
FIND FIRST
FIND SECOND
MAP (A B)
MAP ((C D))
MAP FIND
MAP FIND
(A B)
((C D))
CHECK ONE
CHECK TWO

FIGURE 11

```
    IF the goal is to map a concrete expression to LISP
       and the expression is a data structure involving a term
       and the term corresponds to an argument of the function
THEN the abstract expression can be obtained from the data structure
             by replacing the term with the argument
```

So, in this case she is trying to map the concrete expression (A B) where the argument ONE corresponds to the term (A B). Therefore, after substituting the argument for the term, the abstract expression becomes simply ONE. This same rule applies to map the second concrete expression ((C D)). In this case the argument TWO corresponds to the term (C D) and the abstract expression after substitution is (TWO). Note this rule has correctly mapped the first expression but incorrectly mapped the second expression. The function definition at this point is

```
(def addtwo
  (lambda (one two)
    (cons one (two))))
```

Insert Figure 12 about here

Figure 12 illustrates some of the subsequent evolution of this definition. The coding of ADDTWO had the brother goal of checking that code. Both SS and GRAPES called the LISP interpreter to try the code with the arguments (A B) and (C D). Both received the same error message "TWO undefined function object." This corresponds to an error that SS had encountered a few times previously in her problem solving. In previous occasions, the cause had been failure to quote an argument. Therefore, we assumed that she had compiled an operator that used quote to stop evaluation. When this operator applied her LISP code became

```
(def addtwo
  (lambda (one two)
    (cons one '(two))))
```

Again, the code was tried. This time it returned the result ((A B) TWO). Comparing this with her desired result the problem was localized to the second argument given to CONS and GRAPES went back to retrying the goal of mapping ((C D)).

Insert Figure 13 about here

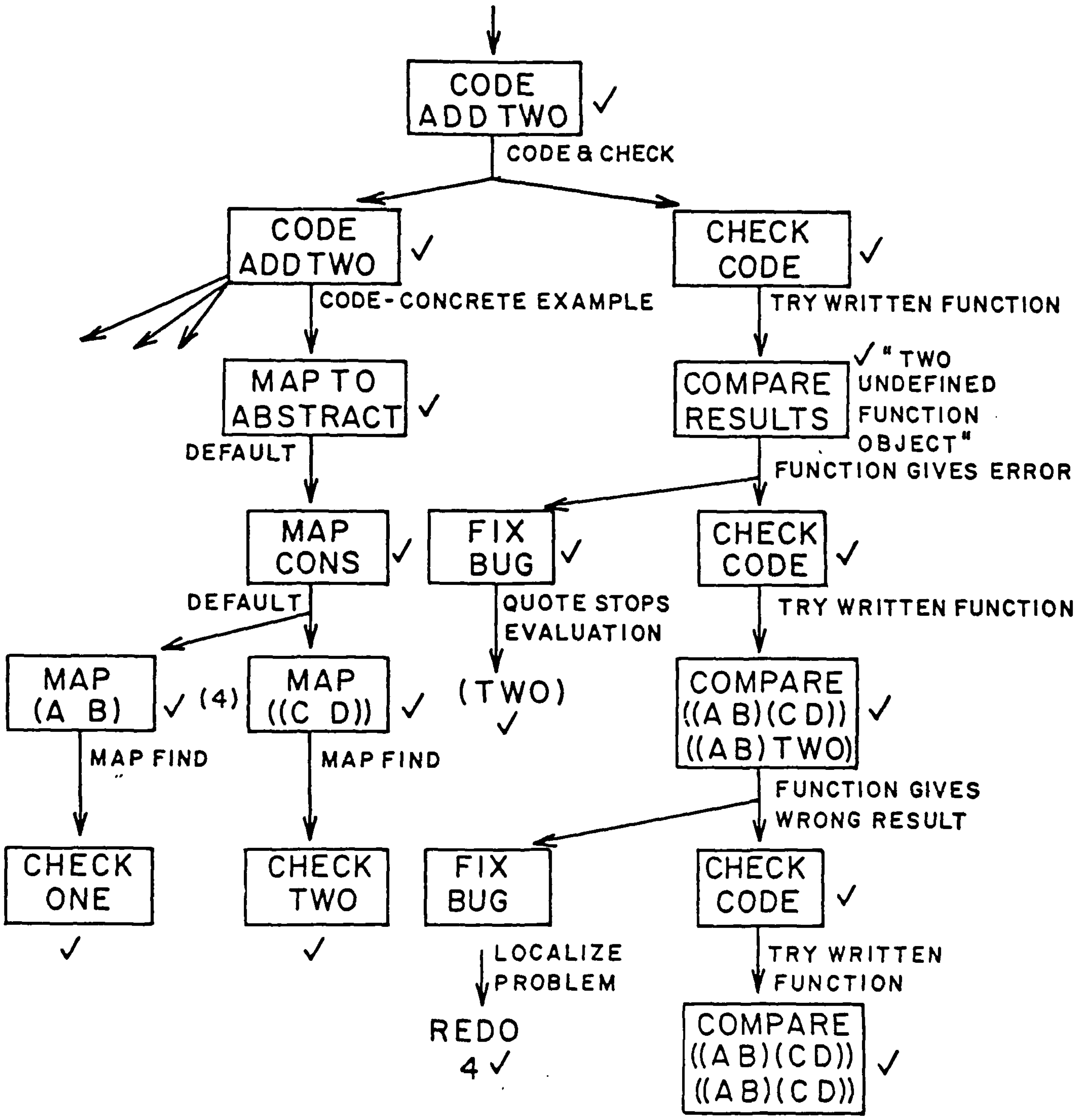

CODE ADD TWO ✓
CODE & CHECK
CODE ADDTWO ✓
CHECK CODE ✓
CODE-CONCRETE EXAMPLE
TRY WRITTEN FUNCTION
MAP TO ABSTRACT ✓
COMPARE RESULTS
✓ "TWO UNDEFINED FUNCTION OBJECT"
DEFAULT
FUNCTION GIVES ERROR
MAP CONS ✓
FIX BUG ✓
CHECK CODE ✓
DEFAULT
QUOTE STOPS EVALUATION
TRY WRITTEN FUNCTION
MAP (A B) ✓ (4)
MAP ((C D)) ✓
(TWO) ✓
COMPARE ((A B)(C D)) ((A B) TWO) ✓
MAP FIND
MAP FIND
FUNCTION GIVES WRONG RESULT
CHECK ONE ✓
CHECK TWO ✓
FIX BUG
CHECK CODE ✓
LOCALIZE PROBLEM
TRY WRITTEN FUNCTION
REDO 4 ✓
COMPARE ((A B)(C D)) ((A B)(C D)) ✓

FIGURE 12

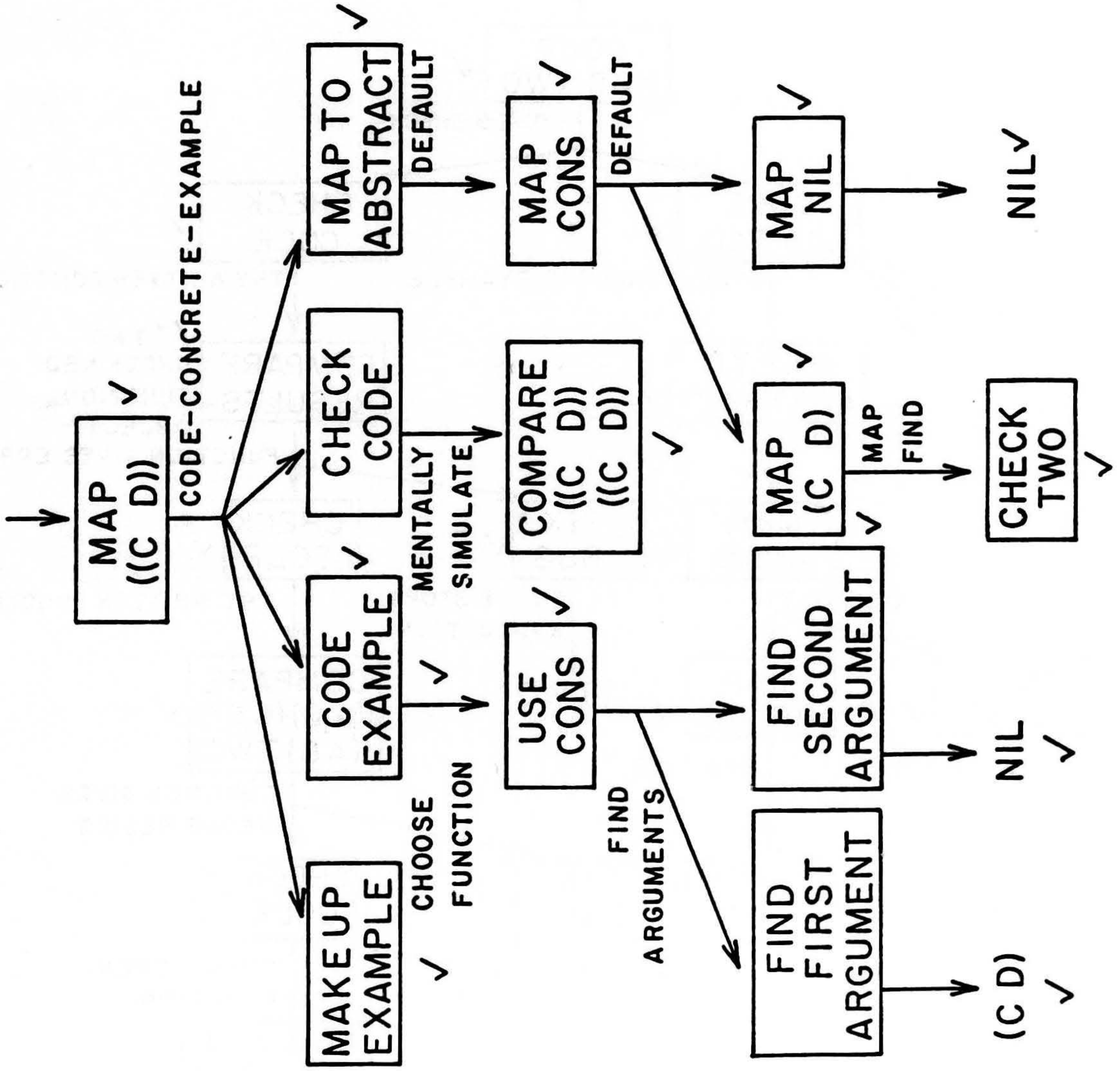

MAP ((C D)) ✓
CODE-CONCRETE-EXAMPLE
MAKEUP EXAMPLE ✓
CODE EXAMPLE ✓
CHECK CODE
MAP TO ABSTRACT ✓
CHOOSE FUNCTION
MENTALLY SIMULATE
DEFAULT
USE CONS ✓
COMPARE ((C D)) ((C D)) ✓
MAP CONS ✓
DEFAULT
FIND ARGUMENTS
FIND FIRST ARGUMENT
FIND SECOND ARGUMENT
MAP (C D) ✓
MAP FIND
MAP NIL ✓
(C D) ✓
NIL ✓
CHECK TWO ✓
NIL ✓

FIGURE 13

Figure 13 illustrates the simulation of this mapping. Having returned to this goal, the previous MAP-FIND operator will not apply again. Therefore, a default rule applies which creates a new subgoal of coding a list consisting of a single argument. As in the case of coding the full ADDTWO problem, GRAPES falls back on the plan of making up a concrete example, coding it, checking the code, and then mapping the code into an abstract code for the function. The previous concrete example of ((C D)) is used. Again, CONS is chosen because it makes lists and again its definition is used to determine the correct arguments. This time the definition is correctly used and GRAPES plans the concrete code as (CONS '(C D) NIL).

After mentally simulating this, GRAPES *turns to the goal of mapping the concrete* code to LISP. The process of performing this mapping is quite analogous to the original mapping in Figure 11. Again, CONS is mapped into CONS. The same MAP-FIND operator as before maps (C D) into TWO. An operator for special LISP symbols, like NIL, maps NIL onto itself. So, the final successful code becomes:

```
(def addtwo
   (lambda (one two)
     (cons one (cons two nil))))
```

One interesting feature of this example is that SS is able to find her way eventually to the correct function without ever correcting the MAP-FIND operator, which will erroneously apply whenever it is given a non-atomic data structure. Later protocols by SS indicated she still had the erroneous MAP-FIND operator. Also, study of novice functions developed for class assignments, suggests that this is a frequent bug. An interesting question concerns the source of this erroneous operator. It is hard to imagine that it was compiled from instruction or example. Presumably it was compiled from an analysis by subjects that data structures should map in a symbol-by-symbol manner, substituting terms in the new domain (in this case, function arguments) for terms in the old domain. If symbols like parentheses are simply treated as default mappings, then one would create an erroneous rule like MAP-FIND. The rule usually works successfully because usually it is given atomic arguments to map. This must make it all the more difficult to eliminate the rule once it gets into the programming repertoire.

4.3. Return to ONETWO

Figure 14 illustrates the behavior of the simulation and the subject when they returned to the original ONETWO problem. The code they generated is given below:

```
(def onetwo
  (lambda (list)
    (cons (first list) (second list))))
```

Whereas the subject had taken an hour to code ADDTWO, she only took ten minutes to solve ONETWO and most of that time was spent confirming what the functions FIRST and SECOND did. ONETWO is solved by the same method that ADDTWO is solved, but without any rehearsal of the ONETWO method. Our assumption is that operators were compiled from this problem that summarized the planning steps that went into the problem solution.

Insert Figure 14 about here

One of the operators that GRAPES compiled summarizes the problem solution illustrated in Figure 13 that started with the goal of creating a list of a single element and resulted in the action of CONSing that element with NIL. The compilation procedure recognizes that the various aspects of the concrete example and its code are intermediate results and are not essential to the final answer. It traces through these steps to determine if there are any connections from the top goal to the final CONSing action. The summary operator built is:

```
    IF the goal is to code a list consisting of one argument
THEN CONS that argument with NIL
        and set as a subgoal to code that argument
```

Similarly, an operator is compiled to correspond to the outer CONS in the ADDTWO function. It has the form

```
    IF the goal is to code into a list consisting of argument1 and argument2
THEN CONS argument1 into a list consisting of argument2
        and set as subgoals to code argument1
        and to code a list consisting of argument2
```

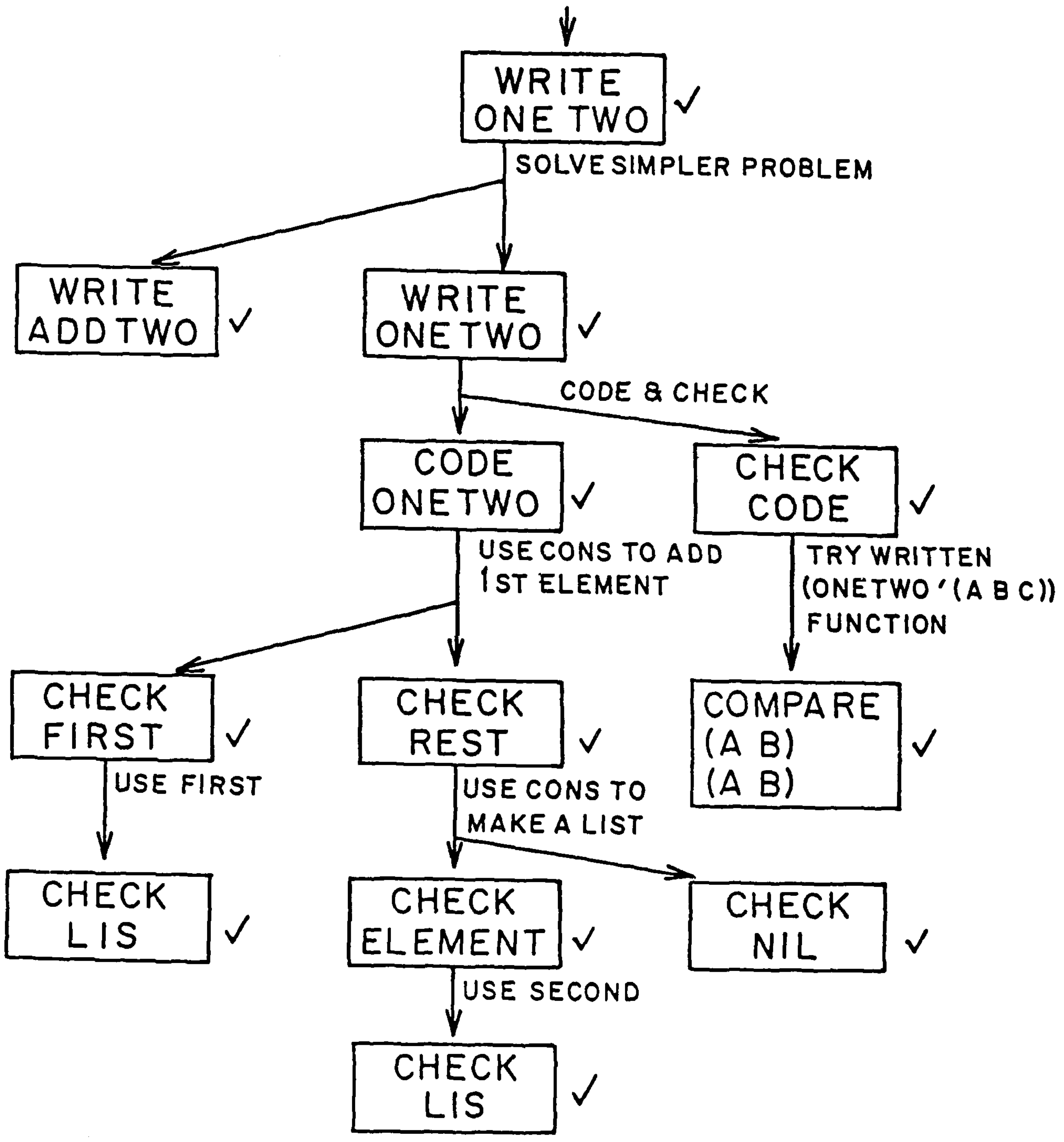

WRITE ONE TWO ✓
SOLVE SIMPLER PROBLEM
WRITE ADDTWO ✓
WRITE ONETWO ✓
CODE & CHECK
CODE ONETWO ✓
CHECK CODE ✓
USE CONS TO ADD 1ST ELEMENT
TRY WRITTEN (ONETWO '(A B C)) FUNCTION
CHECK FIRST ✓
CHECK REST ✓
COMPARE (A B) (A B) ✓
USE FIRST
USE CONS TO MAKE A LIST
CHECK LIS ✓
CHECK ELEMENT ✓
CHECK NIL ✓
USE SECOND
CHECK LIS ✓

FIGURE 14

The compilation operator chooses to work on sections of the goal tree that have the following properties

1. The segment begins with a coding goal such as "code a list of two arguments".
2. The intermediate steps are planning operations whose actions are not essential to the final product.
3. The terminating goals are also concerned with coding.

Compilation produces operators that directly connect 1 to 3 eliminating the intermediate passage through 2.

4.4. Further Discussion of Compilation

As discussed in Anderson (1982) there are two components to compilation-- composition and proceduralization. Composition produces a collapsing of steps and proceduralization eliminates retrieval of information from long-term memory by building that information into the rule. Both components are involved in the compilation of the operators in the ONETWO example, but there are other circumstances where the two might operate singly.

As an example of pure composition, suppose one wanted to add the first member of List1 to List2. Then the following two operators would apply in sequence:

IF the goal is to add an element to List2
THEN CONS the element to List2
and set as subgoals to ~~check~~ the element
and to ~~check~~ List2 code
code

IF the goal is code the first element of List1
THEN use CAR of List1
and set as subgoal to code List1

These two rules could be composed together to produce

IF the goal is to add an element to List2
and the element is the first element of List1
THEN CONS the CAR of List1 with List2
and set as subgoals to code List1
and to code List2

Such composition would collapse repeated sequences of coding operations to create macrooperators. The result would be a speed-up in coding.

Proceduralization can be illustrated in its pure form by the following example: in GRAPES there is a production that will retrieve function definitions from long-term memory and apply them:

```
    IF the goal is to code a relation on an argument list
       and there is a LISP function that codes this relation
THEN  use this function with the argument list
       and set as subgoals to check the coding of the argument list
```

The second line of the condition might match, for instance, "CAR codes the first element of a list." If this rule is proceduralized to eliminate the retrieval of the CAR definition, it becomes

```
    IF the goal is to code the first element of a list
THEN use CAR of the list
       and set as a subgoal to code the list
```

Now a production has been created which can directly recognize the application of CAR. This will result in a reduction in the amount of long-term memory information that needs to be maintained in working memory.

It needs to be emphasized that neither proceduralization nor composition eliminate the original production rules from which they were built. Rather the new compiled rules just serve as additional supplemental rules to produce better performance in certain circumstances.

5. Significant Conclusions

5.1. Types of Rule Formulation

While the following list is probably not exhaustive, we have identified some of the principal types of rules used for programming:

5.1.1. Function Recognition The simplest kind of rule form is one that recognizes an existing LISP function will achieve the current goal. So, for instance, corresponding to the function CONS, there is a production:

```
    IF the goal is to add an element to a list
THEN use CONS
```

```
          and set as subgoals to check the list
          and to check the element
```

These rules result in setting subgoals to check every argument to the function. There are a number of ways that the goal of checking an argument can be achieved. Two of the simplest, which result in direct success, are given by the following rules:

```
        IF the goal is to check an element
          and it corresponds to a variable of the function
   THEN use the variable

        IF the goal is to check an element
          and it has already been coded
   THEN use the existing code.
```

However, if nothing else will work the following default rule will apply.[6]

```
        IF the goal is to check an element
          and the element is defined as having the relation to an argument list
   THEN set as a subgoal to code the relation to the argument list
```

This basically provides a recursion point in GRAPES control structure because we have set a goal of *coding a relation which leads to subgoals of checking* of the arguments to a function that achieves the relation, which lead to new subgoals of finding a function, etc. The simplest and most direct function coding occurs when the problem has been so specified that a set of functions can be hierarchically programmed such that correspond directly to the unpacking of the problem definition. For instance, suppose the following problem were specified, given arguments List1 and List2.

The result is formed by adding X to the front of Y.
X is the first element of List1.
Y is the tail of list List2.

Then the CONS rule would recognize its applicability to obtaining the result and set as subgoals to check X and Y. CAR would be recognized as the code for X and the subgoal would be to check List1. CDR would be recognized as the code for Y and the subgoal would be to check List2. The elements List1 and List2 correspond to the argument variables of the function and the problem would terminate. The resulting code would be (cons (car List1) (cdr List2)). When functions can be coded in this manner, the hierarchical flow of control in GRAPES perfectly reflects the hierarchical structure of LISP.

5.1.2. Problem Refinement One of the reasons why programming is not always as simple as outlined above is that the relations in problem specification may not correspond to known LISP functions. For instance, in the solution of REACHABLE GRAPES comes upon the need to code all the nodes connected to a node. It does not know any LISP code corresponding to the relation <u>connected to</u>. However, it does have the following definition

> The nodes connected to a node appear as the second element of a sublist of graph whose first element is the node.

The following rule can apply

```
        IF the goal is to code a relationship
           and there is a definition of that relationship
      THEN set as subgoals to refine the relationship with the definition
           and then to code the refined relationship
```

In response to the goal to use the definition to refine, the following refined specification of the to-be-coded object is placed in memory.

> It is the second element of a sublist of GRAPH whose first element is the node.

At this point, CADR can recognize its applicability for retrieving the second element and a goal is set to code the sublist of graph whose first element is the node. It is at this point where ASSOC can recognize its applicability.

It needs to be stressed that subjects and GRAPES can fail to recognize the applicability of a function simply because its specification does not match that of the function. Thus, a typical novice "bug" in programming is to write (APPEND (LIST X) Y) rather than (CONS X Y). This is presumably because the subject specifies his goal as adding a list consisting of X to the list Y rather than inserting X as the first element of Y. As another example, if subjects want the sublist of L beginning with element X, they will use (MEMBER X L) but when they want the element after X in L, they might not use (CDR (MEMBER X L)). This is because they do not represent their problem as "the <u>tail of the list beginning</u> with X in L" but rather as "the list after X in L". Thus, the basic point is that programming behavior is not invariant under paraphrase of the problem statement.

5.1.3. Use of Examples and Analogies Generating concrete examples is another method of solving the roadblock of a relation that does to match a known function. We saw this in the ADDTWO protocols. The subject did not know any function directly relevant to putting two elements into a list but generated an example on paper of what the function would do. In parsing this example that she had drawn on the sheet of paper, she noted the result was a list whose first element was the first argument. This matched her definition of CONS.

Another method of solving the roadblock is to solve the problem by hand and try to map the steps of the hand solution to LISP. So, for instance, our subject was asked to write a function that would retrieve the third element of a list, when she only knew CONS, CDR, and CAR. She noted that she was able to solve this by skipping two places in the list and reading the next element. She was then able to translate skipping into taking the CDR and reading the next element into CAR.

The protocol of WC during REACHABLE is another example where a hand solution was used to help guide the structuring of a LISP function. However, sometimes hand solutions do not map naturally into LISP. A good example of this is POWERSET. One of the reasons why POWERSET is so difficult is that subjects' hand solutions do not correspond to the easy LISP solution.

5.1.4. Programming Techniques All the previous methods rely on essentially function definitions, problem definitions, and prior knowledge and skills. However, part of the power in programming comes from acquiring certain problem-solving methods that are specific to programming. POWERSET illustrated the use of tail-recursion or CDR-recursion. This translated the goal of coding the POWERSET relation into the goals of coding the relation between (POWERSET L) and (POWERSET (CDR L)) and of coding the terminating condition. These proved to be much more tractable problems. There are numerous other types of recursion techniques. For instance, there is recursion in the integer N where the problem is decomposed into the goals of coding the relation between (Function N) and (Function N-1) and of coding (Function 0). There are also numerous iteration types which involve their own techniques. There are also various techniques for breaking a problem into subcases and coding each of these separately.

Presumably, it is in the possession of these techniques that the expert programmer is most advanced over the novice. This is an idea that has been suggested by a number of researchers (Kahney & Eisenstadt, 1982; Soloway, 1980; Rich & Shrobe, 1978). Many of these techniques are explicitly learned either through formal courses or informal interaction with other programmers. However, we suspect that many more are also compiled from experience. That is, the programmer hits upon a problem, solves it with much search and effort, and compiles a rule that captures the essence of the solution. There were a couple of modest examples of this in the protocols involving ADDTWO.

5.2. Planning and Top-down Control

The major activity of both our subjects and GRAPES is better characterized as planning rather than coding. Each of the operators involves an attempt to carry out or edt a plan. Some of these operators had code associated with them but that code was just a LISP template attached to the goal structure.

Coding occurs during the program planning and serves to help the subject remember the goal structure of the plan. Thus, this analysis disagrees with the serial analysis of Brooks (1975) who proposed that coding only took place after the plan was complete. The many pages of abortive code written by our subjects is ample evidence that this is not always so.

It is also noteworthy that both our subjects and GRAPES generated this program in a top-down, depth-first, left-to-right manner. The top-down characteristic corresponds to edicts of structured programming; the depth-first characteristic does not. For instance, our subjects and GRAPES wrote the helping function ADDTO for the POWERSET problem before completing the POWERSET function. We think depth-first control is more natural than breadth-first and this is one reason why it is seen in novices.

The basic flow of control in ACT* is depth-first and breadth-first control can only be implemented by embedding it within the depth-first control. For instance, one might imagine implementing a breadth-first queue in a depth-first iterative control which decomposed doing the queue into doing the first element and then doing the rest. For instance,

```
      IF the general strategy is to implement breadth-first expansion
         and the current goal is to execute a queue of goals
    THEN set as a subgoal to do the first goal in the queue
         and set as a second subgoal to do the rest

      IF the general strategy is to implement breadth-first expansion
         and a goal has just led to a set of subgoals
         and there is a queue of subgoals
    THEN append the new goals at the end of the queue
         and set as a subgoal to execute the queue
```

There are a number of reasons why only more expert programmers would show breadth-first control. First, if it has to be implemented in a system that is basically depth-first, then it is more resource-demanding to pursue a breadth-first expansion rather than a depth-first expansion. The novice will have all of his resources taken up in the basics of programming and will not have capacity left over to maintain the breadth-first queue. Second, successful breadth-first expansion requires a sense about what is a reasonable and achievable subgoal. The novice uses depth-first expansion as one means of assessing whether he has just generated a reasonable subgoal. Third, breadth-first expansion becomes more valuable when faced with larger problems. Our subjects never faced a problem that involved more than two helping functions.

An interesting question concerns whether the top-down programming we observed was a result of the hierarchical structure of LISP or whether it would have been observed with other more linear programming languages. Brooks (1975), looking at the coding of FORTRAN, found evidence for a much more linear flow of control although each line of code seemed to be coded in a somewhat top-down manner. Just as it is possible to implement a breadth-first discipline in a depth-first architecture, so is it possible to implement a linear discipline in a top-down architecture or vice-versa. At the current point in time, we cannot say whether the basic cognitive control structures differ in different languages.

Finally, it should be noted that one effect of high levels of knowledge compilation is to make the programming of simple problems appear more linear and less top-down. That is, to the extent that the subject has rules which can recognize large portions of the problem as having known solution, his behavior will be dominated by the linear execution of known code templates rather than hierarchical problem-solving.

5.3. The Role of Analogy

Analogy plays an important role for novices who do not yet have a sufficient set of operators to directly solve the problem. Although not discussed here, a frequent category of analogy involves trying to map one LISP function into another. We discussed two other examples of analogy in this paper. One involves solving a problem in a non-programming domain and then mapping the structure of the solution into LISP code. This was illustrated in our first simulation of the REACHABLE protocol. The other example of analogy was illustrated in the ONETWO simulation. Here the subject tried to solve a problem of finding some code to type in at the top level that had the effect of the function. Then she tried to map this into a LISP function.

It is typically the case that the structure of the old problem in analogy will not be identical to the structure of the new problem to be solved. Therefore, the task of mapping from one to another will not be trivial but rather is a problem-solving task that must be informed by the nature of LISP. While there are exceptions, most of the mapping problems are solved on a trial and error fashion. The effort of our subject trying to map ((C D)) in the ONETWO problem is typical. We believe that students learn how to map analogies to LISP just as they learn other programming skills.

We believe that more expert LISP programmers do not use analogy as frequently because they have compiled out the need for analogy in many situations. Having not looked carefully at expert programmers, this is somewhat a speculation, but it certainly corresponds with our own self observation as LISP programmers. It also is in accord with the GRAPES control structure, which prefers to solve a problem within LISP rather than ~~without~~ outside. As more and more LISP operators are acquired, there will be less need to work outside of LISP.

5.4. Learning by Doing

Text or teacher instruction in LISP does provide the student with valuable facts, but it is only in the course of performing the skill that the student acquires operators that will lead to facile performance of the skill. We have reviewed a number of examples of how the knowledge compilation processes of composition and proceduralization can create new operators. The ACT theory would predict that these operators would become further tuned through generalization and discrimination. However, apparently the first 30 hours of LISP acquisition is too brief to see these tuning processes at work.

5.5. Working Memory

One of the surprise discoveries in our research is how much of the problem-solving protocols are given to recovering from errors of working memory. We estimate that about 50% of the protocols are given to recovering from such memory errors. There are two types of errors observable in the protocols. There are errors that involve forgetting or misremembering subgoals or partial products. Second, there are errors which involve retrieving the wrong item from long-term memory. Typically, this second class of errors involves misremembering function definitions. The most frequent of these is using LIST when CONS is correct. In this paper we gave an analysis of how working memory failure might underlie this memory failure. The proposal was that subjects lost some of the specification of their goal and retrieved a "simpler" function which matched the partial goal specification.

Again, we have not collected systematic data on the matter but we believe that such errors of working memory are less frequent in experts. Certainly, our classroom studies have shown the LIST/CONS confusion decreases with increasing expertise. Part of this improved working memory may result from better problem organization and chunking. However, we also believe that subjects simply have greater working memories for domains and concepts for which they have greater familiarity. Anderson (1983) reviews evidence that more familiar nodes in a semantic network spread greater activation. Chase & Ericsson (1981) showed that with a great deal of practice, subjects can increase their capacity for a list of numbers. They argue that the long-term memories of these subjects become reliable extensions of their short-term memories.

References

Anderson, J.R. Acquisition of cognitive skill. Psychological Review, 1982, 89, 369-406.

Anderson, J.R. The Architecture of Cognition, Harvard University Press, 1983.

Barstow, D.R. An experiment on knowledge-based automatic programming, Artificial Intelligence, 1979, 12, 73-119.

Brooks, R.E. A model of human cognitive behavior in writing code for computer programs. Unpublished doctoral dissertation, Carnegie-mellon University, 1975.

Chase, W. G. and Ericsson, K. A. Skilled memory. In J.R. Anderson (Ed) Cognitive Skills and Their Acquisition. Hillsdale, NJ: Erlbaum, 1981.

Kant, E. and Newell, A. Problem solving techniques for the design of algorithms. To appear in the Proceedings of the Symposium on the empirical foundations of information and software science. Atlanta, GA, November, 1982.

Kahney, H. & Eisenstadt, M. Programmers' mental models of their programming tasks: The interaction of real-word knowledge and programming knowledge. Proceedings of the Fourth Annual Conference of the Cognitive Science Society, 1982.

Miller, G.A., Galanter, E., & Pribram, K.H. Plans and the Structure of Behavior, New York: Holt, 1960.

Norman, D.A. Categorization of action slips. Psychological Review, 1981, 88, 1-15.

Rich, C. & Shrobe, H. Initial report on a LISP programmers' apprentice. IEEE Trans. Soft. Eng., SE-4:6, 1978, 456-466.

Sacerdoti, E.D. A structure for plans and behavior, New York: Elsevier North-Holland, 1977.

Sauers, R. & Farrell, R. GRAPES User's Manual. Technical Report ONR-82-3.

Siklossy, L. Let's Talk LISP, Englewood Cliffs, NJ, 1976.

Soloway, E.M. From problems to programs via plans: The context and structure of knowledge for introductory LISP programming. Coins Technical Report 80-19, University of Massachusetts at Amherst, 1980.

Winston, P.H. Artificial Intelligence. Reading, MA: Addison-Wesley Publishing Company, 1977.

Winston, P.H. & Horn, B.K.P. LISP, Reading, MA: Addison-Wesley, 1981.

Figure Captions

Figure 1	A specification of the REACHABLE problem as presented to subjects.
Figure 2	(a) A illustration of the flow of control in the hand solution of subject WC for the REACHABLE problem. (b) The subject's initial sketch of how this hand solution would result in changes to the list structure representation of the answer.
Figure 3	Hierarchical structure of subject 2's solution to the REACHABLE problem after rule R4 and before the transformation. Checks indicate successful goals and question marks indicate goals yet to be tried.
Figure 4	Hierarchical structure of subject 2's solution to the REACHABLE problem after rules R5-R9 have produced transformation in the goal tree of Figure 3. Checks indicate successful goals and question marks indicate goals yet to be tried.
Figure 5	A specification of the POWERSET problem as presented to subjects.
Figure 6	A representation of the essential insight which underlies solution of the POWERSET problem.
Figure 7	A representation of the hierarchical goal structure controlling GRAPES' solution of the POWERSET problem.
Figure 8	A representation of the hierarchical goal structure controlling GRAPES' solution of the ADDTO problem. This structure is a substructure of the goal structure in Figure 7.
Figure 9	The goal structure at the beginning of the ADDTWO protocol where the subject makes up an example.

This research is supported by contract N00014-81-C-0335 from the Office of Naval Research. We would like to thank Robin Jeffries both for many hours of valuable discussiony relevant to the research and for her ~~copies~~ comments on the paper.

[1]Here and throughout the paper we will give English-like rendition of the production rules. A technical specification of these rules (i.e., a computer listing) can be obtained by writing to us. Also available is a users' manual (Sauers & Farrell, 1982) that describes the system.

[2]Actually, we had a set of productions which generated the hierarchical plan in Figure 2a.

[3]Before these productions apply, a production must apply to refine the definition of connectedness in terms of the GRAPH formalism of the problem. This is discussed in Section 5.1.2.

[4]Unfortunately, this asymmetry is confounded with the fact that 'LIST' is more mnemonic as a function name than CONS. We are currently doing an experiment with artificial function names that attempts to ~~[illegible]~~ eliminate the confound.

[5]We are grateful to discussions with Robin Jeffries for helping our understanding of this problem. Current work with Jeffries is being done on the nature of novice understanding of list structures.

[6]Frequently in the preceeding discussions we have skipped over rules like this which provide the "interstitial" connections between the rules that we were principly interested in.

Navy

1 Dr. Robert Breaux
Code N-711
NAVTRAEQUIPCEN
Orlando, FL 32813

1 CDR Mike Curran
Office of Naval Research
800 N. Quincy St.
Code 270
Arlington, VA 22217

1 DR. PAT FEDERICO
NAVY PERSONNEL R&D CENTER
SAN DIEGO, CA 92152

1 Dr. John Ford
Navy Personnel R&D Center
San Diego, CA 92152

1 LT Steven D. Harris, MSC, USN
Code 6021
Naval Air Development Center
Warminster, Pennsylvania 18974

1 Dr. Jim Hollan
Code 304
Navy Personnel R & D Center
San Diego, CA 92152

1 CDR Charles W. Hutchins
Naval Air Systems Command Hq
AIR-340F
Navy Department
Washington, DC 20361

1 Dr. Norman J. Kerr
Chief of Naval Technical Training
Naval Air Station Memphis (75)
Millington, TN 38054

1 Dr. William L. Maloy
Principal Civilian Advisor for
Education and Training
Naval Training Command, Code 00A
Pensacola, FL 32508

Navy

1 CAPT Richard L. Martin, USN
Prospective Commanding Officer
USS Carl Vinson (CVN-70)
Newport News Shipbuilding and Drydock Co
Newport News, VA 23607

1 Dr William Montague
Navy Personnel R&D Center
San Diego, CA 92152

1 Ted M. I. Yellen
Technical Information Office, Code 201
NAVY PERSONNEL R&D CENTER
SAN DIEGO, CA 92152

1 Library, Code P201L
Navy Personnel R&D Center
San Diego, CA 92152

1 Technical Director
Navy Personnel R&D Center
San Diego, CA 92152

6 Commanding Officer
Naval Research Laboratory
Code 2627
Washington, DC 20390

1 Psychologist
ONR Branch Office
Bldg 114, Section D
666 Summer Street
Boston, MA 02210

1 Office of Naval Research
Code 437
800 N. Quincy SStreet
Arlington, VA 22217

5 Personnel & Training Research Programs
(Code 458)
Office of Naval Research
Arlington, VA 22217

1 Psychologist
ONR Branch Office
1030 East Green Street
Pasadena, CA 91101

Navy

1 Special Asst. for Education and
Training (OP-01E)
Rm. 2705 Arlington Annex
Washington, DC 20370

1 Office of the Chief of Naval Operations
Research Development & Studies Branch
(OP-115)
Washington, DC 20350

1 LT Frank C. Petho, MSC, USN (Ph.D)
Selection and Training Research Division
Human Performance Sciences Dept.
Naval Aerospace Medical Research Laborat
Pensacola, FL 32508

1 Dr. Gary Poock
Operations Research Department
Code 55PK
Naval Postgraduate School
Monterey, CA 93940

1 Dr. Worth Scanland, Director
Research, Development, Test & Evaluation
N-5
Naval Education and Training Command
NAS, Pensacola, FL 32508

1 Dr. Alfred F. Smode
Training Analysis & Evaluation Group
(TAEG)
Dept. of the Navy
Orlando, FL 32813

1 Dr. Richard Sorensen
Navy Personnel R&D Center
San Diego, CA 92152

1 Roger Weissinger-Baylon
Department of Administrative Sciences
Naval Postgraduate School
Monterey, CA 93940

1 Dr. Robert Wisher
Code 309
Navy Personnel R&D Center
San Diego, CA 92152

Navy

1 Mr John H. Wolfe
Code P310
U. S. Navy Personnel Research and
Development Center
San Diego, CA 92152

Army

1 Technical Director
U. S. Army Research Institute for the
Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Mr. James Baker
Systems Manning Technical Area
Army Research Institute
5001 Eisenhower Ave.
Alexandria, VA 22333

1 Dr. Beatrice J. Farr
U. S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 DR. FRANK J. HARRIS
U.S. ARMY RESEARCH INSTITUTE
5001 EISENHOWER AVENUE
ALEXANDRIA, VA 22333

1 Dr. Michael Kaplan
U.S. ARMY RESEARCH INSTITUTE
5001 EISENHOWER AVENUE
ALEXANDRIA, VA 22333

1 Dr. Milton S. Katz
Training Technical Area
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Harold F. O'Neil, Jr.
Attn: PERI-OK
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Robert Sasmor
U. S. Army Research Institute for the
Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Joseph Ward
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Air Force

1 Dr. Earl A. Alluisi
HQ, AFHRL (AFSC)
Brooks AFB, TX 78235

1 Dr. Alfred R. Fregly
AFOSR/NL, Bldg. 410]
Bolling AFB
Washington, DC 20332

1 Dr. Genevieve Haddad
Program Manager
Life Sciences Directorate
AFOSR
Bolling AFB, DC 20332

2 3700 TCHTW/TTGH Stop 32
Sheppard AFB, TX 76311

Marines

1 H. William Greenup
Education Advisor (E031)
Education Center, MCDEC
Quantico, VA 22134

1 Special Assistant for Marine
Corps Matters
Code 100M
Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217

1 DR. A.L. SLAFKOSKY
SCIENTIFIC ADVISOR (CODE RD-1)
HQ, U.S. MARINE CORPS
WASHINGTON, DC 20380

CoastGuard

1 Chief, Psychological Reserch Branch
U. S. Coast Guard (G-P-1/2/TP42)
Washington, DC 20593

Other DoD

12 Defense Technical Information Center
Cameron Station, Bldg 5
Alexandria, VA 22314
Attn: TC

1 Military Assistant for Training and
Personnel Technology
Office of the Under Secretary of Defense
for Research & Engineering
Room 3D129, The Pentagon
Washington, DC 20301

1 DARPA
1400 Wilson Blvd.
Arlington, VA 22209

Civil Govt

1 Dr. Paul G. Chapin
Linguistics Program
National Science Foundation
Washington, DC 20550

1 Dr. Susan Chipman
Learning and Development
National Institute of Education
1200 19th Street NW
Washington, DC 20208

1 Dr. John Mays
National Institute of Education
1200 19th Street NW
Washington, DC 20208

1 William J. McLaurin
66610 Howie Court
Camp Springs, MD 20031

1 Dr. Arthur Melmed
National Intitute of Education
1200 19th Street NW
Washington, DC 20208

1 Dr. Andrew R. Molnar
Science Education Dev.
and Research
National Science Foundation
Washington, DC 20550

1 Dr. Joseph Psotka
National Institute of Education
1200 19th St. NW
Washington,DC 20208

1 Dr. Frank Withrow
U. S. Office of Education
400 Maryland Ave. SW
Washington, DC 20202

1 Dr. Joseph L. Young, Director
Memory & Cognitive Processes
National Science Foundation
Washington, DC 20550

Non Govt

1 Anderson, Thomas H., Ph.D.
Center for the Study of Reading
174 Children's Research Center
51 Gerty Drive
Champiagn, IL 61820

1 Dr. John Annett
Department of Psychology
University of Warwick
Coventry CV4 7AL
ENGLAND

1 1 psychological research unit
Dept. of Defense (Army Office)
Campbell Park Offices
Canberra ACT 2600, Australia

1 Dr. Alan Baddeley
Medical Research Council
Applied Psychology Unit
15 Chaucer Road
Cambridge CB2 2EF
ENGLAND

1 Dr. Patricia Baggett
Department of Psychology
University of Colorado
Boulder, CO 80309

1 Dr. Jonathan Baron
Dept. of Psychology
University of Pennsylvania
3813-15 Walnut St. T-3
Philadlphia, PA 19104

1 Mr Avron Barr
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Liaison Scientists
Office of Naval Research,
Branch Office , London
Box 39 FPO New York 09510

1 Dr. Lyle Bourne
Department of Psychology
University of Colorado
Boulder, CO 80309

Non Govt

1 DR. JOHN F. BROCK
Honeywell Systems & Research Center
(MN 17-2318)
2600 Ridgeway Parkway
Minneapolis, MN 55413

1 Dr. John S. Brown
XEROX Palo Alto Research Center
3333 Coyote Road
Palo Alto, CA 94304

1 Dr. Bruce Buchanan
Department of Computer Science
Stanford University
Stanford, CA 94305

1 DR. C. VICTOR BUNDERSON
WICAT INC.
UNIVERSITY PLAZA, SUITE 10
1160 SO. STATE ST.
OREM, UT 84057

1 Dr. Pat Carpenter
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

1 Dr. John B. Carroll
Psychometric Lab
Univ. of No. Carolina
Davie Hall 013A
Chapel Hill, NC 27514

1 Dr. William Chase
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

1 Dr. Micheline Chi
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. William Clancey
Department of Computer Science
Stanford University
Stanford, CA 94305

Non Govt

1 Dr. Allan M. Collins
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, Ma 02138

1 Dr. Lynn A. Cooper
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. Meredith P. Crawford
American Psychological Association
1200 17th Street, N.W.
Washington, DC 20036

1 Dr. Kenneth B. Cross
Anacapa Sciences, Inc.
P.O. Drawer Q
Santa Barbara, CA 93102

1 LCOL J. C. Eggenberger
DIRECTORATE OF PERSONNEL APPLIED RESEARC
NATIONAL DEFENCE HQ
101 COLONEL BY DRIVE
OTTAWA, CANADA K1A OK2

1 Dr. Ed Feigenbaum
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Mr. Wallace Feurzeig
Bolt Beranek & Newman, Inc.
50 Moulton St.
Cambridge, MA 02138

1 Dr. Victor Fields
Dept. of Psychology
Montgomery College
Rockville, MD 20850

1 Univ. Prof. Dr. Gerhard Fischer
Liebiggasse 5/3
A 1010 Vienna
AUSTRIA

Non Govt

1 Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, MA 02138

1 Dr. Alinda Friedman
Department of Psychology
University of Alberta
Edmonton, Alberta
CANADA T6G 2E9

1 Dr. R. Edward Geiselman
Department of Psychology
University of California
Los Angeles, CA 90024

1 DR. ROBERT GLASER
LRDC
UNIVERSITY OF PITTSBURGH
3939 O'HARA STREET
PITTSBURGH, PA 15213

1 Dr. Marvin D. Glock
217 Stone Hall
Cornell University
Ithaca, NY 14853

1 Dr. Daniel Gopher
Industrial & Management Engineering
Technion-Israel Institute of Technology
Haifa
ISRAEL

1 DR. JAMES G. GREENO
LRDC
UNIVERSITY OF PITTSBURGH
3939 O'HARA STREET
PITTSBURGH, PA 15213

1 Dr. Harold Hawkins
Department of Psychology
University of Oregon
Eugene OR 97403

1 Dr. Barbara Hayes-Roth
The Rand Corporation
1700 Main Street
Santa Monica, CA 90406

Non Govt

1 Dr. Frederick Hayes-Roth
The Rand Corporation
1700 Main Street
Santa Monica, CA 90406

1 Dr. Dustin H. Heuston
Wicat, Inc.
Box 986
Orem, UT 84057

1 Dr. James R. Hoffman
Department of Psychology
University of Delaware
Newark, DE 19711

1 Dr. Kristina Hooper
Clark Kerr Hall
University of California
Santa Cruz, CA 95060

1 Glenda Greenwald, Ed.
"Human Intelligence Newsletter"
P. O. Box 1163
Birmingham, MI 48012

1 Dr. Earl Hunt
Dept. of Psychology
University of Washington
Seattle, WA 98105

1 Dr. Ed Hutchins
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Greg Kearsley
HumRRO
300 N. Washington Street
Alexandria, VA 22314

1 Dr. Steven W. Keele
Dept. of Psychology
University of Oregon
Eugene, OR 97403

1 Dr. Walter Kintsch
Department of Psychology
University of Colorado
Boulder, CO 80302

Non Govt

1 Dr. David Kieras
Department of Psychology
University of Arizona
Tuscon, AZ 85721

1 Dr. Stephen Kosslyn
Harvard University
Department of Psychology
33 Kirkland Street
Cambridge, MA 02138

1 Dr. Marcy Lansman
Department of Psychology, NI 25
University of Washington
Seattle, WA 98195

1 Dr. Jill Larkin
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

1 Dr. Alan Lesgold
Learning R&D Center
University of Pittsburgh
Pittsburgh, PA 15260

1 Dr. Michael Levine
Department of Educational Psychology
210 Education Bldg.
University of Illinois
Champaign, IL 61801

1 Dr. Mark Miller
TI Computer Science Lab
C/O 2824 Winterplace Circle
Plano, TX 75075

1 Dr. Allen Munro
Behavioral Technology Laboratories
1845 Elena Ave., Fourth Floor
Redondo Beach, CA 90277

1 Dr. Donald A Norman
Dept. of Psychology C-009
Univ. of California, San Diego
La Jolla, CA 92093

Non Govt

1 Committee on Human Factors
JH 811
2101 Constitution Ave. NW
Washington, DC 20418

1 Dr. Seymour A. Papert
Massachusetts Institute of Technology
Artificial Intelligence Lab
545 Technology Square
Cambridge, MA 02139

1 Dr. James A. Paulson
Portland State University
P.O. Box 751
Portland, OR 97207

1 Dr. James W. Pellegrino
University of California,
Santa Barbara
Dept. of Psychology
Santa Barabara, CA 93106

1 MR. LUIGI PETRULLO
2431 N. EDGEWOOD STREET
ARLINGTON, VA 22207

1 Dr. Richard A. Pollak
Director, Special Projects
Minnesota Educational Computing Consorti
2520 Broadway Drive
St. Paul,MN 55113

1 Dr. Martha Polson
Department of Psychology
Campus Box 346
University of Colorado
Boulder, CO 80309

1 DR. PETER POLSON
DEPT. OF PSYCHOLOGY
UNIVERSITY OF COLORADO
BOULDER, CO 80309

1 Dr. Steven E. Poltrock
Department of Psychology
University of Denver
Denver,CO 80208

Non Govt

1 Dr. Mike Posner
Department of Psychology
University of Oregon
Eugene OR 97403

1 MINRAT M. L. RAUCH
P II 4
BUNDESMINISTERIUM DER VERTEIDIGUNG
POSTFACH 1328
D-53 BONN 1, GERMANY

1 Dr. Fred Reif
SESAME
c/o Physics Department
University of California
Berkely, CA 94720

1 Dr. Lauren Resnick
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Mary Riley
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. Andrew M. Rose
American Institutes for Research
1055 Thomas Jefferson St. NW
Washington, DC 20007

1 Dr. Ernst Z. Rothkopf
Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

1 Dr. David Rumelhart
Center for Human Information Processing
Univ. of California, San Diego
La Jolla, CA 92093

1 DR. WALTER SCHNEIDER
DEPT. OF PSYCHOLOGY
UNIVERSITY OF ILLINOIS
CHAMPAIGN, IL 61820

Non Govt

1 Dr. Alan Schoenfeld
Department of Mathematics
Hamilton College
Clinton, NY 13323

1 DR. ROBERT J. SEIDEL
INSTRUCTIONAL TECHNOLOGY GROUP
HUMRRO
300 N. WASHINGTON ST.
ALEXANDRIA, VA 22314

1 Committee on Cognitive Research
% Dr. Lonnie R. Sherrod
Social Science Research Council
605 Third Avenue
New York, NY 10016

1 Dr. David Shucard
Brain Sciences Labs
National Jewish Hospital Research Center
National Asthma Center
Denver, CO 80206

1 Robert S. Siegler
Associate Professor
Carnegie-Mellon University
Department of Psychology
Schenley Park
Pittsburgh, PA 15213

1 Dr. Edward E. Smith
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

1 Dr. Robert Smith
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

1 Dr. Richard Snow
School of Education
Stanford University
Stanford, CA 94305

1 Dr. Kathryn T. Spoehr
Pscyhology Department
Brown University
Providence, RI 02912

Non Govt

1 Dr. Robert Sternberg
Dept. of Psychology
Yale University
Box 11A, Yale Station
New Haven, CT 06520

1 DR. ALBERT STEVENS
BOLT BERANEK & NEWMAN, INC.
50 MOULTON STREET
CAMBRIDGE, MA 02138

1 David E. Stone, Ph.D.
Hazeltine Corporation
7680 Old Springhouse Road
McLean, VA 22102

1 DR. PATRICK SUPPES
INSTITUTE FOR MATHEMATICAL STUDIES IN
THE SOCIAL SCIENCES
STANFORD UNIVERSITY
STANFORD, CA 94305

1 Dr. Kikumi Tatsuoka
Computer Based Education Research
Laboratory
252 Engineering Research Laboratory
University of Illinois
Urbana, IL 61801

1 Dr. John Thomas
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

1 DR. PERRY THORNDYKE
THE RAND CORPORATION
1700 MAIN STREET
SANTA MONICA, CA 90406

1 Dr. Douglas Towne
Univ. of So. California
Behavioral Technology Labs
1845 S. Elena Ave.
Redondo Beach, CA 90277

1 Dr. Benton J. Underwood
Dept. of Psychology
Northwestern University
Evanston, IL 60201

Non Govt

1 DR. GERSHON WELTMAN
PERCEPTRONICS INC.
6271 VARIEL AVE.
WOODLAND HILLS, CA 91367

1 Dr. Keith T. Wescourt
Information Sciences Dept.
The Rand Corporation
1700 Main St.
Santa Monica, CA 90406

1 DR. SUSAN E. WHITELY
PSYCHOLOGY DEPARTMENT
UNIVERSITY OF KANSAS
LAWRENCE, KANSAS 66044

1 Dr. Christopher Wickens
Department of Psychology
University of Illinois
Champaign, IL 61820

1 Frank R. Yekovich
School of Education
Catholic University